

Motorola Semiconductor Application Note

AN461

An Introduction to the HC16 for HC11 Users

By **Ross Mitchell**
MCU Applications Group
Motorola Ltd., East Kilbride, Scotland

Introduction — Basic Design Philosophy of the M68HC16

The M68HC16 (HC16) is a highly modular device family based on the CPU16 16-bit core. The CPU16 core is a true 16-bit design, with an architecture that will be very familiar to M68HC11 (HC11) users. The resemblances to the HC11 core design are a deliberate move to provide an upgrade path for those 8-bit 68HC11 designs that require the increased power of a 16-bit CPU. Many features of the HC16 and the CPU16 core will be new to HC11 users, and it is these changed and new features that this document aims at explaining.

The HC16 provides a software upgrade path for HC11 users while giving full hardware compatibility with the asynchronous address and data bus found on the 32-bit microprocessors.

The basic HC11 CPU is easily recognizable in [Figure 4](#), with a number of additional registers enhancing the flexibility of the core. The addition of the multiply-accumulate (MAC) block ([Figure 7](#)) provides the user with greatly improved digital signal processing (DSP) capabilities. Many architectural changes have been made to improve the performance of the CPU.

The hardware interface has, however, been radically changed. This is now compatible with the asynchronous address/data bus interface found

on the 68000, 68020, and 68300 Families of devices. To greatly reduce the external logic, a module called the system integration module (SIM) has been designed and provides the signals required to control the external bus.

The HC16 is built with the 0.8-micron double metal HCMOS (high-density metal-oxide semiconductor) process.

The following discusses the various differences likely to be encountered by a user of the M68HC16 who is experienced in using the HC11. It covers the CPU architecture, software compatibility, and hardware of the HC16 device. A detailed table of contents follows. Since there are a number of technical references available, it will be assumed the reader has these to hand.

Table of Contents

Introduction — Basic Design Philosophy of the M68HC16	1
The Concept of the Intermodule Bus (IMB)	7
Explanation of the Basic IMB Concept	7
Basic Starting Position	9
Modularity	9
Choosing a Module List for IMB Designs	11
On-Chip Peripherals	11
CPU16	12
System Integration Module (SIM)	12
General-Purpose Timer Module (GPT)	13
Standby RAM (SRAM)	14
Serial Communications Interface (SCI)	15
Serial Peripheral Interface (SPI)	16
Analog-to-Digital Converter (ADC)	18
Ports	18

Basic CPU16 Core Architecture Differences	19
A Look at the Non-DSP Parts of the Core for the CPU16 . . .	19
CCR Register	22
K Registers	24
Program and Data Space	25
Examination of the DSP Part of the CPU16 Core	28
HC16 n-Tap FIR Filter	29
Use of SM, EV, and MV Bits of the CCR	31
Source Code Compatibility	31
Basic Approach to Source Code Compatibility	31
Changes in Detail	32
Addressing Modes	32
Timing Changes	34
Assembly Code Differences Between HC11 and HC16	34
Enhancements in CPU16 Source Code	36
20-Bit Addressing	36
Moving Data without Affecting the Accumulators	36
MAC and RMAC	38
Stack Operations	40
Difference between RTI and RTS	42
16-Bit Signed Branch	43
Pipelining	43
16- and 32-Bit Arithmetic	44
Comparison of HC11 and HC16 Code and Benchmarks . . .	47
Check List of Changes to HC11 Code	47
Initialization of HC16 Device	48
PLL Control	48
SRAM Initialization Procedure	49
Stack Initialization	50
K Register Initialization	50
Mode Selection During Reset	52
Reset Operation	52
Vectors, Stack Operations	54
Exception Routine Address	54
User-Defined Vectors	55
New Vectors for HC11 Users	55
Reset Status Register	56

Exception Handling (Interrupts)	59
Interrupt Request Handling	59
Module Design Influence on the Conversion from HC11 to HC16 Code	59
Setting Up an Internal Exception	59
HC11 Timer Initialization	60
HC16 Code for GPT Interrupt Initialization	61
Initializing the QSPI	63
Setting Up an External Exception	63
Periodic Interrupt vs. Real-Time Interrupt	65
Different Exception Levels	66
Arbitration	67
Same Exception Level	68
Multiple Exception Events	68
Prioritization Schemes	70
Exception Routine Entry Latency	70
External Hardware Interfacing	73
Asynchronous vs. Synchronous Bus	73
Wait States	74
Fast Termination (Synchronous Timing)	75
Using Chip Selects	75
8-Bit and 16-Bit Read/Write Access to 8-Bit Wide Memory Devices	75
Hardware for 8-Bit and 16-Bit Addressing Using a Single-Chip Select	77
V _{DDE} vs. V _{DDI} vs. V _{DDA}	78
Minimum Required Connections for the SIM	79
Debugging Tools	83
Background Mode	83
Evaluation Board	83
Appendix A	84
Appendix B	85
Appendix C	86

Figures, Tables, and Examples

Figure	Title	Page
1	68HC16Z1, 68HC16Y1, and 68332 Block Diagrams	8
2	68HC11E9 and 68HC11K4 Block Diagrams	8
3	QSPI RAM Model	17
4	HC11 Register Set	19
5	Basic HC16 Register Set Minus the MAC Registers	21
6	HC11 and CPU16 Condition Code Registers	24
7	MAC Registers	28
8	MAC Instruction Flow	29
9	MAC Instruction Operation	39
10	Schematic Diagram of the PLL	48
11	Multiple Interrupts	69
12	8- and 16-Bit Address Read and Write Access with CSBOOT	78
13	Shows the Suggested Decoupling as Close to the HC16 Pins as Possible for These Pairs of Power Pins . .	79
14	HC11 with External Memory	81
15	HC16 with External Memory	82
16	MC68HC11E9 Device	84
17	MC68HC16Z1 Device	85
18	Simplified HC16 Timing Diagram	86

Table	Title	Page
1	Initial Modules Available for the IMB Family	10
2	Example Baud Rates Possible with a System Clock of 16.78 MHz	15
3	Accumulator D and E Instructions Compared	23
4	HC11 Instructions Modified for CPU16 Implementation	35
5	Move Instruction	36
6	DSP Register Initialization	38
7	DSP Support Instructions	40
8	HC11 Stack Control Instructions	41
9	CPU16 Stack Control Instructions	42
10	Registers that Must/Should be Written after Reset	51
11	List of 1-Time Write Bits/Registers	52
12	SIM Configuration Out of Reset	53
13	Vector Table Definition for the HC16	57
14	Generating an Autovector and Initializing the Watchdog	64
15	Periodic Interval Timer Setup and Exception Handler	66

Example	Title	Page
1	HC16 Code with Data Accesses the Same 64-Kbyte Segment	26
2	HC16 Code Data Accesses Across Adjacent 64-Kbyte Segments	27
3	HC11 Code to Average 64 10-Bit A/D Values	30
4	N-Tap FIR Filter for the HC16	30
5	HC16 Moving Data from an ADC Result Register to a RAM Table	37
6	HC11 Code to Calculate 16-Bit Times 8 Bits Divided by 16 Bits	45
7	HC16 Code to Perform 16-Bit Times 8-Bit Divided by 16 Bits	45
8	HC11 (8-Bit x 8-Bit x 8-Bit) / 9-Bit	46
9	HC16 (8-Bit x 8-Bit x 8-Bit) / 9-Bit	46
10	MC68HC16Z1 Initialization Routine	51
11	HC11 Initialization Code	51
12	Definition of HC11 Vector Table	56
13	Actual HC16 Vector Table	58
14	HC11 Code for Timer Initialization	60
15	HC16 GPT Initialization for Interrupts on OC2	62
16	Initialization of the QSPI	63
17	HC11 Timer Output Compare 2 Interrupt Routine	71
18	HC16 Timer Output Compare 2 Exception Handler	72
19	Initialization Code for 8- and 16-Bit Addressing of External Memories	76

The Concept of the Intermodule Bus (IMB)

Explanation of the Basic IMB Concept

The HC11 device is made up of a number of functionally different modules which are connected together to form a fully operational microcontroller. These modules range from CPU (central processor unit) and ROM (read-only memory) to very complex timers and communications interfaces. The HC16 employs the same techniques, but goes one step further by standardizing the shape and interface of the modules to one another. Each module must be designed so that there is the absolute minimum of change in connection to the rest of the design, and this means that the intermodule connections and the external pin connections are standardized.

This is achieved by the intermodule bus (IMB) which is a standard bus interface for all internal modules of the HC16 and 68300 Families of microcontrollers. It consists of 16 data lines, 24 address lines, and numerous control lines that are available to all modules in the device.

In general terms, the IMB has a similar function to an address and data bus on any computer system. This bus is very like the 68020 asynchronous bus and so uses handshaking between the sending and receiving modules to allow a data transfer to occur. Thus, a very large number of modules can be accommodated simply with little design effort for each variant of the HC16 or 68300.

The system integration module (SIM) is simply the logic required to run the asynchronous address and data bus and takes care of both internal and external bus activity with little differentiation between them. It also provides control of interrupt events and includes a number of systemwide functions such as system monitors and clock generation.

One thing that you will notice is that the IMB concept results in a “back bone” for the device, with a visible track stretching across the middle of the device. See [Figure 1](#) showing the block diagrams of the HC16 devices 68HC16Z1 and 68HC16Y1 for an example of the back bone effect. Compare these to the design of the 68HC11E9 and 68HC11K4 devices in [Figure 2](#) and note the irregular shapes of the HC11 devices.

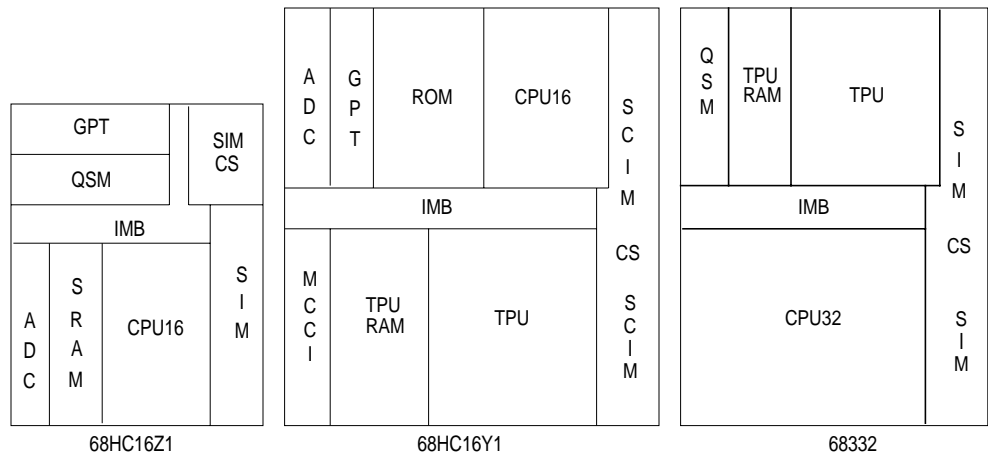


Figure 1. 68HC16Z1, 68HC16Y1, and 68332 Block Diagrams

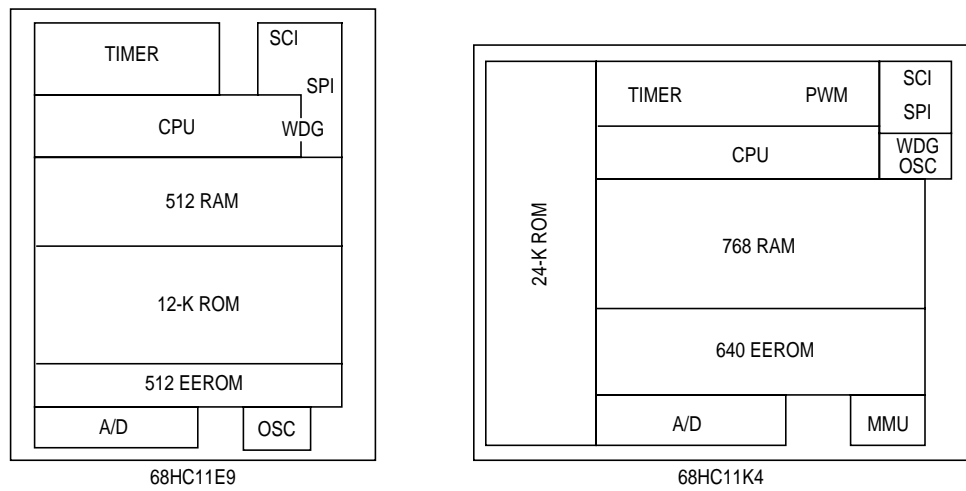


Figure 2. 68HC11E9 and 68HC11K4 Block Diagrams

Basic Starting Position

A couple of modules always exist on an intermodule bus (IMB) device. The IMB itself is, of course, absolutely required. A CPU is a good idea, and then one must choose a system integration module to suit the application. After that, there is a free hand in choosing of the modules.

Modularity

The HC11 and HC05 customer will be well acquainted with the CSIC (customer-specified integrated circuit) approach. This is simply where the customer specifies a custom integrated circuit, and it will often have a completely new module that must be designed to support the customer's particular needs. Where a design does not involve a new module to be designed, the complete design and layout process can take as little as three months. Most of this time is spent integrating the customer-specified modules and incorporating them into a rectangular piece of silicon, where each module is often an irregular shape.

The intermodule bus (IMB) allows even faster integration by specifying a standard shape and interconnectivity for all modules. The IMB lies along the center of the device and each module has a fixed height such that only the width of the silicon will vary according to the number and width of each module. [Figure 1](#) illustrates two devices in the HC16 Family and compares them with the 68332, the first CPU32-based IMB device.

It is a relatively simple matter to assemble the appropriate modules to give the best utilization of the silicon, and then all that remains is to connect the external pads of the device to the modules. As a consequence, a 3- to 6-month design period can turn out vastly more complex devices than the HC11 and HC05 in a comparable time frame.

The modularity of each module normally requires that there be no customization of the module so that it has precisely the same functionality as that module on another device. This is not a restriction as the modules themselves are therefore designed to provide the maximum flexibility for the user. As new requirements of the MCUs are demanded, the library of modules increases. Currently, 12 are available and many more are in design. See [Table 1](#) for a brief list of the first modules to become available.

A clear result of this modular approach is that the initialization of each module is very important and certainly more complex than would be the case for an HC11 device. In the majority of cases, the registers are pre-initialized to a value that would be either safe or expected in normal operation.

The HC11 offers an approach called the fixed chassis customer-specified integrated circuit (FCCSIC), where the basic device chassis and pinout are restricted to a few combinations and there is an area of silicon known as white space which is available for any random logic design. There is no such equivalent as yet for the IMB Family, but often the flexibility of the modules will account for much of the logic required in these cases.

Table 1. Initial Modules Available for the IMB Family

CPU16	16-bit CPU, based on HC11 core
CPU32	32-bit CPU, based on 68020 core
SIM	System integration module (chip selects, clock, system protection)
SCIM	Single-chip SIM (Address/data bus can become port lines.)
SRAM	Static RAM with low-voltage standby operation
TPURAM	TPU emulation SRAM
TPU	Timer processor unit, 16-channel timer with RISC-type core
EEROM	16-k and 48-k FLASH EEROM modules
ROM	48-Kbyte ROM module
QSM	SCI and queued SPI module
MCCI	2 SCIs and a single SPI
ADC	8/10-bit, 8-channel A to D converter

Choosing a Module List for IMB Designs

Making a choice of modules is a complex task. In essence, it comes down to choosing the functionality required of the application and building up the device out of the available blocks. Since the functionality of each module is very specific, the application can often be broken down to match these functions. The choice is then between two or three modules and becomes much more obvious.

Each module has its own reference manual and so it is easy to collect all the relevant data on each module function for analysis of a system's design of an application.

As the Family of HC16 devices grows, it is increasingly likely that the user will find a device close to the ideal requirements of the application. This makes the task of defining a new variant of the HC16 much simpler, since the combination of critical modules can be assessed easily.

On-Chip Peripherals

The HC16 consists of a number of different modules, each with its own specialist function which will combine to give the functionality of a powerful microcontroller. The CPU and the external bus interface are obvious elements of such a device and are known as CPU16 and the system integration module (SIM), in this case. These are connected to the IMB just as any other module would and so it is important to consider the functions of the core and the SIM as separate from one another, with communication via the intermodule bus (IMB).

Much of this document is directed at giving the reader a basic understanding of the features of the CPU16 and SIM modules since these are the basic blocks of any version of the HC16. Many other modules exist today, with many more in design. The following, therefore, will concentrate on the peripherals found on the MC68HC16Z1, since this device most closely resembles the very popular MC68HC11E9 device.

CPU16

The core is a module like all other modules. It is a true 16-bit CPU with some novel features. It has a pseudo-linear address capability of 1 Mbyte for the user program, while the data space is built up from 16 banks or segments of 64 Kbytes. The CPU16 also boasts a digital signal processor (DSP) functionality with a full MAC capability, allowing 16-bit x 16-bit multiply and addition into a 36-bit wide accumulator in a single 720-ns instruction.

The programmer's model can be seen in [Figure 4](#), [Figure 5](#), and [Figure 6](#).

System Integration Module (SIM)

The system integration module is one of the most complex modules available and interfaces between the internal device modules and the external peripheral devices on the system. It has 12 chip selects, 24 address lines, 16 data lines, seven interrupt pins, numerous bus control lines, digital I/O (input/output) ports, a periodic interval timer, arbitration logic, system monitors, and the system clock generation via a PLL (phase-locked loop).

The chip selects are a large piece of the SIM and are basically 12 comparators that can check the 24 bits of the address bus and some of the bus control signals such as address strobe, read/write, and address space type. This then allows a handshake signal called DSACK to be signalled to the external bus interface section of the SIM, and this in turn completes the bus cycle. The comparator is also used for generation of AVEC, a signal to indicate the presence of an autovector (interrupt event).

The bus monitor is disabled after reset, but when enabled it will flag (BERR) a long delay in the bus cycle completion, possibly due to a write or read from non-existent memory.

The PLL allows a 32.78-kHz crystal to be connected to the external oscillator pins and then multiply up the frequency by the use of a prescaler in the feedback circuit of the PLL from the VCO to the phase detector. It is this multiplied clock that becomes the system clock. The prescaler is a 6-bit modulus prescaler with separate divide-by-2 and divide-by-4 selections and so can generate a system clock ranging from 131 kHz up to 16.78 MHz from a 32.78-kHz input clock. This allows the

user to arrange the exact frequency for timers, periodic timer, SCI, etc., to suit the application. See [Figure 10](#) for a block diagram of the PLL.

A significant portion of this document aims to explain the operation of the SIM. The basic blocks of the SIM and its external pins can be seen in [Appendix B](#).

The following is a very brief overview of the HC11-like peripherals found on the HC16 device. These peripherals offer many improved features over the HC11 equivalent, and so the user must refer to the appropriate reference manuals to take full advantage of the increased functionality.

General-Purpose Timer Module (GPT)

The general-purpose timer (GPT) module is based mainly on the HC11F1 or HC11E9 timer. It has three input captures, four output compares, an input capture or output compare channel, a pulse accumulator input, and two 8-bit PWM (pulse-width modulation) outputs and a PWM clock input. The PWM is similar to the 68HC05B6's PWM.

The differences are:

1. Maximum of 4 MHz (240 ns) timer clock (2-MHz HC11 has a maximum of 500-kHz (2 μ s) timer clock)
2. The pulse accumulator input is a separate pin from the OC1 channel.
3. The PWMs are additional. These are more sophisticated than the HC05B6 PWMs, since they have a prescaler in addition to the fast/slow mode. They do not, however, have the modulus counters of the HC11G5 device.

The two 8-bit PWMs can run from a maximum of 32.78 kHz down to just 4 Hz. These cannot, however, be concatenated as is the case with the HC11G5.

The PWM clock can come from an external pin called PCLK.

4. All the input capture and output compare pins have alternate functions of digital I/O capability, and the PWM pins may be used as discrete output pins.

Standby RAM (SRAM)

The SRAM can be mapped anywhere in the 1-Mbyte addressable program or data space. To move the SRAM, the STOP (RAM enable/disable) bit must be set, meaning that the RAM is disabled, and the RLCK (RAM base address lock) bit must not have been written to a 1 state. (This is a write-once register bit.) A write to the RAM base address registers RAMBAR (\$FFB04) and (\$FFB06) will allow the user to place the RAM base address anywhere in memory.

A good place to put the SRAM is at address \$F0000, since this can be accessed by the same instructions that access the module registers as they, too, are situated in bank 15.

The SRAM can be either program or program and data space. This allows the user to decide whether code can be run out of RAM or if it is to be used for data storage.

If using the RAM only for program space, the initialization is important. Since there is no way to write data from the CPU registers to program space, it is necessary to initialize the RAM as program and data space first and then copy the user code into RAM. The RASP (RAM array space) bit is then set and the RAM will only execute code from RAM until the RASP bit is cleared once more.

An application for running code from RAM would possibly be where the user has slow or maybe 8-bit wide external memory in the system and wishes to use the internal 16-bit wide fast termination RAM for execution of a critical piece of code that requires short cycle times. Remember that in program space-only mode there is no possibility of storing data in RAM from the CPU registers.

NOTE: *The SRAM is disabled in standby mode which operates whenever the V_{STBY} pin is approximately 0.5 volts higher than the V_{DDI} supply. To avoid this situation, the V_{STBY} pin may be grounded and the SRAM will always be powered from V_{DDI} .*

Serial
Communications
Interface (SCI)

The SCI and the queued SPI share the QSM module and so the registers will all be found together.

The SCI operates in almost exactly the same way as the SCI of the HC11K4. This means that it has a modulus register to generate the baud rate for the receive and transmit and so can have a very wide range of baud rates to allow for the variable system clock frequency possible from the PLL.

The HC11E9 has a much simpler SCI baud rate generator that has fixed divide ratios that sometimes cause problems for users wishing to run the device at 3-MHz bus speed and still require 9600 baud, for example. With the modulus baud rate divider, a simple equation is used to generate the appropriate baud rate. It is:

$$\text{Baud rate} = \text{system clock} / (32 * \text{BR})$$

where BR is the divide ratio selected in the baud rate register (\$FFC08).

Table 2. Example Baud Rates Possible with a System Clock of 16.78 MHz

Nominal Baud Rate	Actual Baud Rate	Percent Error	Baud Rate Register Value
500,000	524,288	+4.9	1
38,400	37,449	-2.5	14
19,200	19,418	+1.1	27
9600	9533	-0.7	55
4800	4810	+0.2	109
2400	2405	+0.2	218
1200	1200	0	437
300	300	0	1748
110	110	0	4766

Clearly, the PLL can be used to bring these small errors to a minimum.

The other features of the SCI that have changed are relatively minor. The SCI has an option for wired OR mode for the TXD output, and it will automatically generate the parity bit and check the parity on receiving data. Also, there is a bit in the status register called RAF that is set when the SCI receiver is busy, indicating that another message is being received.

Serial Peripheral Interface (SPI)

The basic SPI from the HC11 is recognizable after a good long look at the QSPI. There are a great many additional features in this module which have been grafted onto the HC11 type of SPI. It should be a relatively easy job to convert the HC11 over to the QSPI function, but it is more likely that the user will wish to take advantage of the improved functionality of this module in particular to reduce unnecessary CPU intervention in the SPI transfers.

The SPI is a queued SPI and so has a 16-word transmit and 16-word receive storage RAM within the module. The actual function of the SPI remains unchanged, but each of the 16 words of transmit/receive RAM are controlled by an 8-bit command register that autonomously controls the SPI operation without direct control from the CPU.

The queue can send or receive a number of words or bytes and can manipulate the external slave select lines continuously or in a burst and then await further intervention from the CPU.

The obvious applications are for control of an SPI LCD display or external A/D converter, to name just two external peripherals. Here there is a need to send or receive several bytes/words of data regularly and the QSPI can handle this with little or no CPU control once started. In the case of the A/D, the data could be read from the receive queue by an interrupt service routine and the SPI would just continue to collect the data automatically.

The queue control byte associated with each of the 16 sets of receive and transmit registers is made up from two 4-bit fields. The first field controls the protocol and the second field determines which of the external SPI devices is being accessed.

Thus, it is possible to determine the number of bytes transferred, whether there is a delay after a transfer, the delay from slave select to transmission start, and whether to disable the slave devices between data transfers on each SPI transfer. Also, the four slave select lines can be changed on each of the 16 transfers.

In addition to the number of bytes, the QSPI can transmit/receive any number of bits from eight up to 16, thus making communication with a 10-bit external A/D much simpler.

The SPI baud rate definition is very similar to that for the SCI baud rate. Again, there is a modulus prescaler, set by the 8-bit BAUD field of the SPCR0 (QSPI control register 0) register at \$FFC18. The equation is set as follows:

$$\text{Baud rate} = \text{system clock} / (2 * \text{baud})$$

where baud has values ranging from 2 up to 256.

At reset, the baud register is preset to 4, giving a baud rate of 2.1 MHz with a 16.78-MHz system clock. The maximum baud rate is 4.19 MHz and the minimum baud rate is 33 kHz (with the 16.78-MHz system clock).

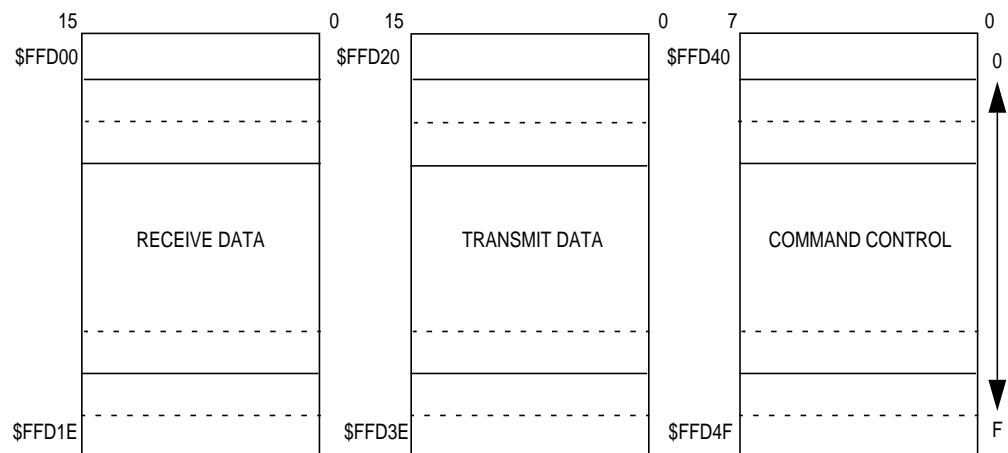


Figure 3. QSPI RAM Model

Analog-to-Digital Converter (ADC)

The ADC is basically the same as the MC68HC11E9 in its functionality, but it does have a number of significant improvements.

For example, the module can operate as an 8-bit or 10-bit converter (taking 1 μ s longer to convert a 10-bit result). The results of all eight channels can be read from their individual data registers in three different data representations: right justified unsigned, left justified unsigned, and left justified signed. This means that there are a total of 24 16-bit wide registers for the eight channels of 10-bit A/D. The data formatting is intended to make the application of DSP functions much simpler by providing the results in signed form. The left justified unsigned value gives easy access to the most significant 8-bit result, while the right justified unsigned value gives easy handling of the 10-bit result.

Conversion times are 8 μ s for eight bits and 9 μ s for 10 bits.

The other ADC features — such as continuous sampling, single conversion and stop, and grouping of four or eight channels — are much the same as for the HC11G5.

Ports

Each module on the HC16 has its own digital port lines as an alternate function to the primary module function. The GPT has a parallel digital I/O port which operates in precisely the same way as port A on the HC11.

Since the modules are designed to be completely separate, the port line control registers are scattered across the register/address map. In some cases, the individual control of port pins may be in completely different registers. An example of this is the timer processor unit (TPU), where the digital I/O pin control comes from the control for each timer channel.

Basic CPU16 Core Architecture Differences

A Look at the Non-DSP Parts of the Core for the CPU16

A close inspection of the CPU16 register set in [Figure 5](#) reveals very close similarity to the corresponding registers of the HC11 in [Figure 4](#). There are several additional registers, with accumulator E, index register IZ largely being duplicates of the accumulator D and index register IY, respectively. The K register is completely new, the result of extending the address range of the CPU. The condition code register has changed quite a bit to support the enhancements to the CPU.

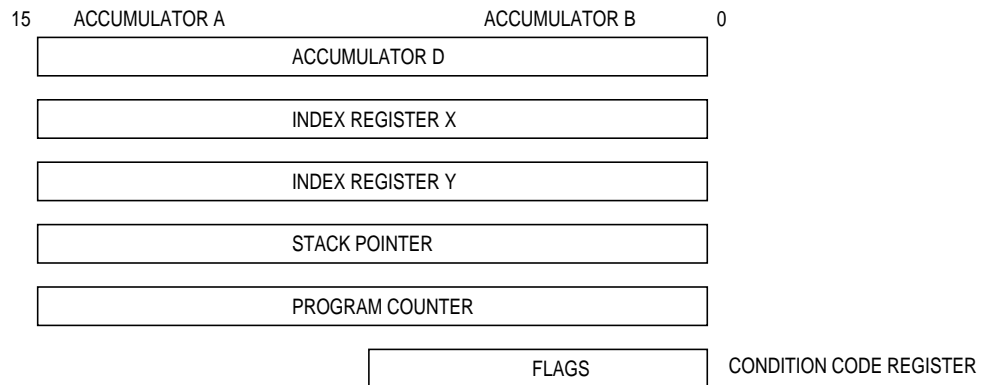


Figure 4. HC11 Register Set

Accumulators D and E differ slightly from one another. Accumulator E is only a 16-bit accumulator (unlike accumulator D, which also can be considered as two 8-bit accumulators). As can be seen in [Table 3](#), most of the instructions apply equally to accumulators D and E. All the logical, comparative, and data movement instructions are identical for both accumulators; however, there are some differences with the arithmetic instructions. Just as with the HC11, there is a decrement and increment on the 8-bit registers A and B, but since there is no 16-bit decrement on the HC11, there is no such instruction on the CPU16. Accumulator E, therefore, has no associated decrement/increment instruction. Additionally, the ABA instruction is complemented by the ADE instruction on the CPU16; however, there is no instruction to add E to D.

NOTE: *The decimal adjust only operates on accumulator A. In a similar way, reading from/writing to the CCR is possible only from accumulators A or D for an 8-bit or 16-bit operation, respectively.*

The implications of this is to continue to concentrate on accumulator D as the primary 16-bit accumulator and use accumulator E for the results of the calculations. All accumulator D and accumulator E instructions take the same number of clock cycles to execute. This is further emphasized by the addition of two new types of instruction for the CPU16. These are the 16-bit offset on an indexed instruction and the accumulator E offset on an indexed instruction.

There is only a 16-bit signed offset available for the accumulator E instructions whereas accumulator D has the possibility of an unsigned 8-bit offset or a signed 16-bit offset on the index register. The 8-bit offset is directly compatible with the HC11 and is normally a 2-byte opcode, making it a very efficient and normally 2-cycle (120 ns) instruction.

The accumulator E offset on an index register is a significant improvement over the HC11. This will allow an offset calculation of a signed 16-bit value to be used to directly access a memory location. This allows the index register to remain pointing at the top of the data field rather than save the pointer address and then restore later as required by the HC11. [Example 1](#) and [Example 2](#) show how this instruction might be used.

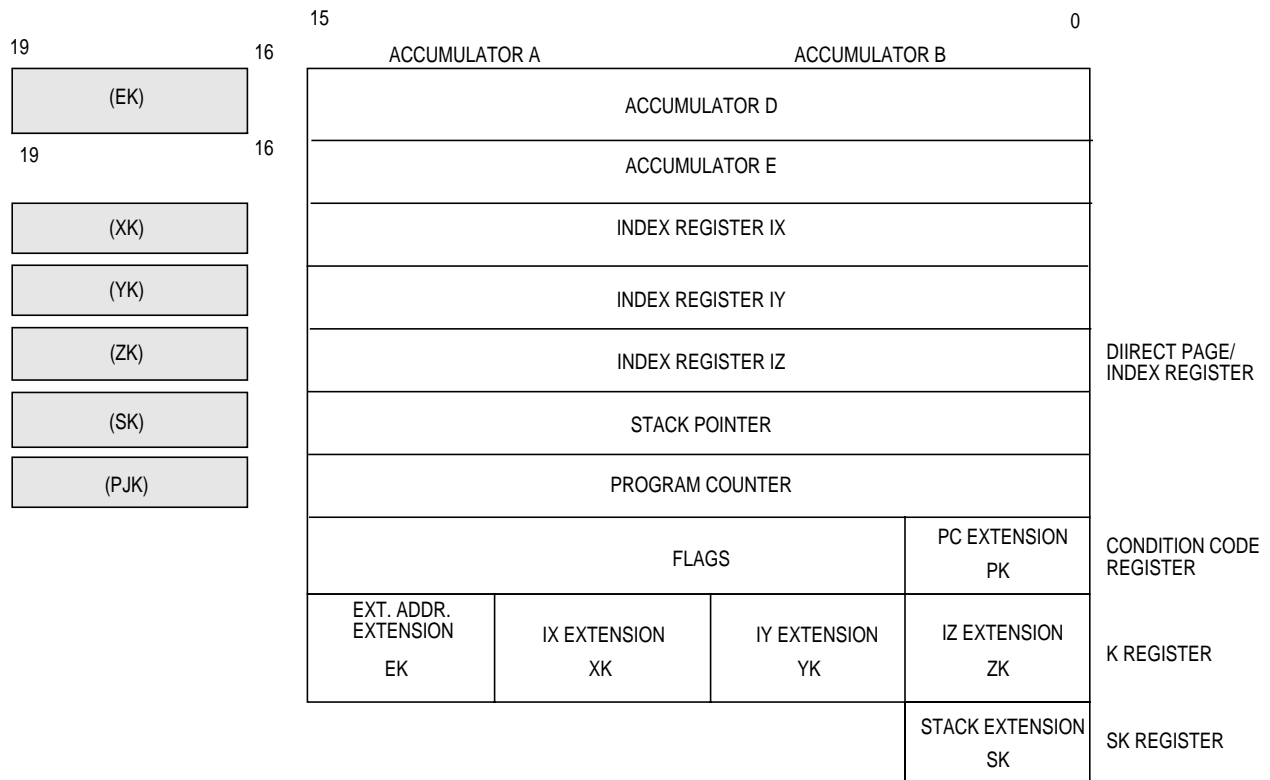


Figure 5. Basic HC16 Register Set Minus the MAC Registers

Table 3 also shows that there are 32-bit instructions. These are primarily intended for use with 32-bit registers found on some modules for the IMB bus. In particular, the TPU can be configured to have 32-bit registers which must be written in one memory access cycle. See **Example 5** for an example of this instruction in a program example. As is shown later in this section, there are also special uses for accumulator E when using the multiply accumulate functions in the CPU16.

CCR Register

At first sight, there is little difference between the CCR (condition code register) of the HC11 and that of the HC16. Most of the bits from the HC11 are included in the CPU16 CCR, but there are a number of changes (see [Figure 6](#)). Bits C, V, Z, N, H, and S operate the same way for both devices. The I and X bits in the HC11 CCR are replaced by three bits: I0, I1, and I2. This is as a result of a major change in the approach to interrupt handling on the HC16. Rather than have a single non-maskable interrupt and a prioritized maskable interrupt, the HC16 has seven levels of interrupt or exception mask, the highest level (7) of which is non-maskable. The section on [Exception Handling \(Interrupts\)](#) covers this in detail.

NOTE: *After reset, the interrupt mask is at level 7.*

CPU16 bits SM, EV, and MV relate to the MAC instruction and are covered later in this section. The PK field is the 4-bit extension to the PC register to allow 20-bit addressing by the CPU16 program counter. When the CCR and PC are stacked during an interrupt, there is no need for a special additional stack operation for the PK register. During a branch to subroutine, the HC11 would normally stack just the PC, but here we need the additional four bits of the PK register and so both the CCR and PC registers are also stacked for a branch-to-subroutine operation on CPU16. An RTS instruction discards the stacked CCR content on returning to the calling routine.

Table 3. Accumulator D and E Instructions Compared

Instruction	Accumulator A 8-Bit	Accumulator B 8-Bit	Accumulator D 16-Bit	Accumulator E 16-Bit	Accumulator E: Accumulator D 32-Bit
ARITHMETIC add with carry add add B to A add D to E subtract with carry subtract subtract B from A subtract D from E decrement increment decimal adjust clear	ADCA ADDA ABA ADE SBCA SUBA SBA SDE DECA INCA DAA CLRA	ADCB ADDB SBCB SUBB DECB INCB CLRB	ADCD ADDD SBCD SUBD CLR D	ADCE ADDE SBCE SUBE CLRE	
LOGICAL complement and negate OR exclusive OR arithmetic shift L arithmetic shift R logical shift R rotate left rotate right	COMA ANDA NEGA ORAA EORA ASLA ASRA LSRA ROLA RORA	COMB ANDB NEGB ORAB EORB ASLB ASRB LSRB ROLB RORB	COMD ANDD NEGD ORD EORD ASLD ASRD LSRD ROLD RORD	COME ANDE NEGE ORE EORE ASLE ASRE LSRE ROLE RORE	
COMPARATIVE compare test	CMPA TSTA	CMPB TSTB	CPD TSTD	CPE TSTE	
DATA MOVEMENT load store transfer exchange stack unstack	LDAA STAA TBA XGAB PSHA PULA	LDAB STAB TAB XGAB PSHB PULB	LDD STD TED XGDE PSHM D PULM D	LDE STE TDE XGDE PSHM E PULM E	LDED STED PSHM D,E PULM D,E
CCR OPERATIONS transfer A to CCR transfer CCR to A transfer D to CCR transfer CCR to D	TAP TPA		TDP TPD		
INDEX REGISTER CONTROL add to index X add to index Y add to index Z			ADX ADY ADZ	AEX AEY AEZ	

Figure 5 shows the K registers both in their symbolic location and in their apparent physical location. As can be seen, the index K registers and the EK register are grouped together in a single 16-bit wide register called, unsurprisingly enough, K. PK is added to the lowest nibble of the CCR. This makes a great deal of sense when it comes to stacking the CPU registers for interrupt and subroutines. Clearly, SK should not be altered after a stacking operation and so need not be accessible in a 16-bit wide register for stacking purposes.

The K registers are the most obvious core differences to an HC11 user, but they are normally set up at the start of a program and after that, rarely altered. This is a good idea, not least because a new value for the 4-bit K registers must first be loaded in the accumulator B and then transferred to nK (nK is XK or YK or ZK or EK or SK) by the command TBnK and read from nK by the command TnKB. A primary reason for not changing the K registers is that they normally point to data either within the program or generated by the program or HC16 peripheral modules. Often, the K registers may take advantage of the CPU distinguishing between data and program space.

Program and Data Space

NOTE: *There is a difference between data and program address space for the CPU16. The CPU determines that any access of data that requires the use of the program counter (and, of course, the PK register) must be an address space known as program space. Data space is any memory access via the accumulators (such as a read or write operation with accumulator E). A third type of space is CPU space and this will be explained later in the section concerning hardware interfacing and external interrupts.*

Program space is a linear 1-Mbyte addressable space and the CPU16 automatically handles all the manipulation of the additional 4-bit PK register. Thus, there is no way to manipulate the PC or PK registers other than with stack operations.

Data space is 16 segments of 64 Kbytes of address space. The EK, XK, YK, ZK, and SK extension registers all point to data space. The CPU will not automatically change the K register contents when a series of data instructions crosses a 64-Kbyte boundary (unlike the PK register for program space). The user must, therefore, ensure that the tables of data are within a 64-Kbyte segment or take care of the boundary conditions as the access to the data fields cross the 64-Kbyte boundary.

The following examples illustrate the small modifications required to ensure data accesses will operate between adjacent segments. It only takes five additional lines of code and 10 bytes to make **Example 1** work over a bank boundary. The resultant code can be found in **Example 2**. Both examples perform the same task, but **Example 2** has no restrictions on where the input table lies.

Another interesting point is the use of the std, e,y instruction. Had the output table in RAM gone across a bank boundary, this code would still work because accumulator E is added to YK:IY and the value of YK will be temporarily incremented if the sum of accE and IY exceeds \$FFFF. YK itself is never changed by such an instance.

Example 1. HC16 Code with Data Accesses the Same 64-Kbyte Segment

```

***** MOVE the data within a bank boundary *****
* 64-Kbyte ROM extends from $0000 to $0FFFF
* Move 256 words of table from ROM to internal RAM in bank 15
*
RAM_TABLE    equ    $0040            ; output table at $F0040 (RAM in bank 15)
TABLE        equ    $F000            ; input table starts at $0F000 (bank 0)
TABLE_END    equ    $F200            ; input table ends at $0F200 (bank 0)
TABLE_LENGTH equ    $200             ; table is 512 bytes in length

                ldab #$0              ; point to bank 0 for ROM table
                tbxk
                ldab #$F              ; point to bank 15 for RAM
                tbyk
                lde  #$0000
                ldy  #RAM_TABLE       ; destination in SRAM
                ldx  #TABLE           ; source in ROM
moveloop
                ldd  0,x              ; get the data from ROM
                std  e,y              ; write it to RAM
                aix  #2               ; increment source pointer by a word
                adde #2               ; increment destination pointer by a word
                cpe  #TABLE_LENGTH    ; check if past end of table
                blt  moveloop         ; if not then continue to move data

```

Example 2. HC16 Code Data Accesses Across Adjacent 64-Kbyte Segments

```

***** MOVE the data within a bank boundary *****
* 128-Kbyte ROM extends from $0000 to $1FFFF
* Move 256 words of table from ROM to internal RAM in bank 15
*
RAM_TABLE      equ    $0040          ; output table at $F0040 (RAM in bank 15)
TABLE          equ    $FF80          ; input table starts at $0FF80 (bank 0)
TABLE_END      equ    $0180          ; input table ends at $10180 (bank 0)
TABLE_LENGTH   equ    $200           ; table is 512 bytes in length

                ldab   #0             ;
                tbxk                ; point to bank 0 (ROM) for input table
                ldab   #$F           ;
                tbyk                ; point to bank 15 (RAM) for output table
                lde    #$0000
                ldy    #RAM_TABLE     ; destination in SRAM
                ldx    #TABLE         ; source in ROM

moveloop
                ldd    0,x            ; get the data from ROM
                std    e,y            ; write it to RAM
                aix    #2             ; increment source pointer by a word

check_bank
                cpx    #$0000         ; check if data runs over a bank boundary
                bne    same_bank      ; if IX not $0000 then still in same bank

new_bank
                txkb                ; if IX=$0000 then get the current bank
                addb   #1             ; increment it by 1
                tbsk                ; update the bank pointer

same_bank
                adde   #2             ; increment destination pointer by a word
                cpe    #TABLE_LENGTH ; check if past end of table
                blt    moveloop       ; if not then continue to move data
  
```

Examination of the DSP Part of the CPU16 Core

A totally new function of the core is the multiply-accumulate capability. As can be seen from [Figure 7](#), there are a number of new registers that are intended to be used to multiply two 16-bit numbers pointed to by index registers X and Y and the result added to the contents of a 36-bit register called M by a single instruction MAC (multiply-accumulate). This instruction can post increment/decrement up to 15 both the X and Y index registers to allow a series of MAC n m commands to multiply and accumulate the results of a series of values very quickly. This function is for digital filtering and it can complete a MAC in 720 ns (12 clock cycles).

The H and I registers are used as pointers to the indexed data and the XMASK and YMASK bytes control the actual addressing of the data. These masks, when not set to 0, will effectively give an upper limit on the address range of the H and I registers. This gives a wrap around effect to the data, but remember to start the table of input data and coefficient data at a multiple of the X and Y mask value. The table size is a power of 2 of the XMASK maximum bit set plus 1. A value of XMASK equal to \$1 will allow a table of 2 values, while XMASK equal to \$3 will allow four values and XMASK equal to \$7 will allow a table of eight values.

DSP algorithms can be found in many HC11 applications and the CPU16 core will perform the same function very much faster.

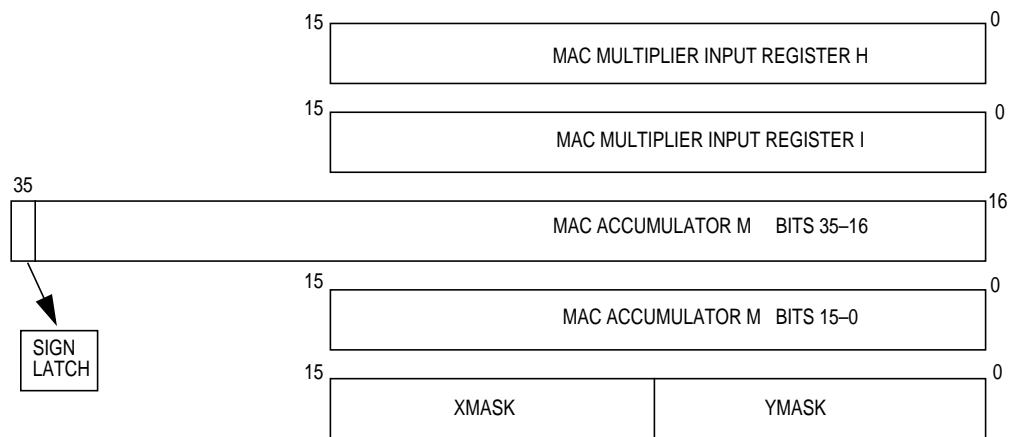


Figure 7. MAC Registers

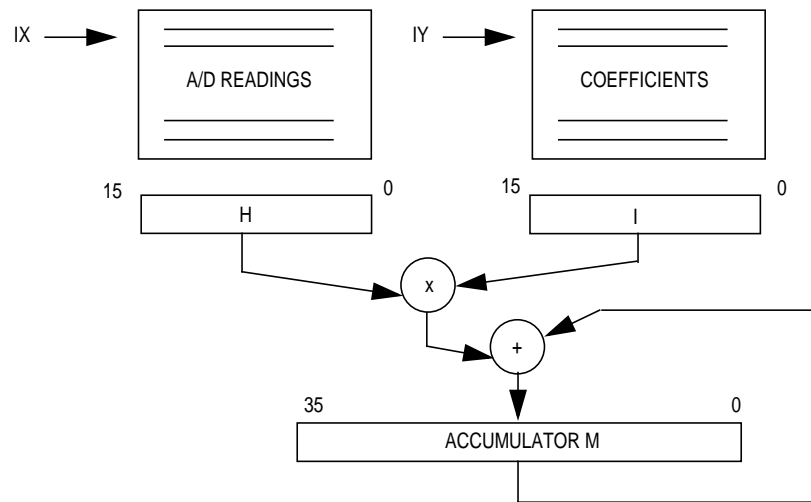


Figure 8. MAC Instruction Flow

HC16 n-Tap FIR Filter

It is often found that the HC11 device was required to perform some averaging of data to reduce noise in the A/D reading, for example. Normally, this would simply be a buffer with the last n readings from the A/D channel and this would be updated regularly, while a separate routine would add up all the results and divide the answer by the number of readings averaged. In effect, this is making the HC11 perform a low-pass digital filter function.

The HC16 device has the MAC and RMAC instructions to allow it to perform more than just an averaging function on the incoming data. To show the actual operation of the DSP part of the core, the following shows a simple FIR filter performing the averaging routine through the use of what is called a finite response filter (FIR).

The 64 results to be averaged, it would be necessary for the HC11 to divide the sum of the values by 64. If the average is more than a 10-bit result, then the HC11 would have problems storing the sum of the results because this would exceed 16 bits. A solution to this is to divide each result by 64 or a multiple of 2 at least to prevent overflow, but this would reduce the accuracy of the result. The only alternative is to perform arithmetic on greater than 16 bits which is time consuming for the HC11.

*Use of SM, EV,
and MV Bits
of the CCR*

EV saturation of the M accumulator shows that there has been an overflow into bit 31. The value of M is now greater than 0.99997 or less than -0.99997.

MV extension of the M accumulator shows that there has been an overflow into bit 35 and that the result in accumulator M is incorrect and changed from a positive number to a negative number. This bit is written to an invisible register called the signlatch which will retain the sign of the overflow.

SM (saturation mode) is a bit that can be written in the CCR. When the MV bit or EV bit is set, the result in accM (31:16) will be either the most positive (\$7FFF) or most negative (\$8000) number, depending upon the sign of the overflow. This simulates analogue saturation.

Source Code Compatibility

Basic Approach to Source Code Compatibility

The HC16 is considered to be source code compatible with the HC11. This means that the user can take HC11 code and pass it through an HC16 assembler and it will function just as before (ignoring the obvious hardware differences). With a few exceptions, which will be explained in this section, this is indeed the case.

The CPU16 core is fully 16-bit and has the additional feature of 20-bit addressing built into the core design. This accounts for many of the changes, but these new features do not normally affect the way that HC11 source code would operate. Interrupts, stack operations, and communication with on-chip peripherals are the main areas of change, and these are very simple to make.

Changes in Detail

In fact, a great deal of effort has gone into maintaining HC11 source code compatibility. For example, there is now an exception vector for a divide by 0; however, this does not function with the HC11 command IDIV or FDIV. It will function only with the new CPU16 divide instruction, EDIV and EDIVS. Similarly, there is a direct page (page zero) addressing mode to enable the user to use code efficient 8-bit offset operations anywhere in memory.

At a quick glance, HC16 assembly code looks very much like HC11 code. The most obvious difference is in the initialization sequence at the start of the user program. Here, the K registers are set up and the peripheral modules configured for the specific requirements of the application. Since the HC11 can only address 64 Kbytes (except, of course, the HC11K4), the K registers most likely will be set to 0 or F.

K reg = 0 will point to the lowest 64-Kbyte address space where the reset vectors are situated and where the user program most likely will start. The registers for the peripherals, however, are located at the very top of the 1-Mbyte memory address. Therefore, they can be accessed either by forcing the read/write at the top 64-Kbyte segment or by a negative offset on the address.

Addressing Modes

The CPU16 core makes full use of signed arithmetic and so it is important to remember that an unsigned operation on the HC11 could become a signed operation with the HC16. The 16-bit indirect offset is an unsigned value on the HC11, but becomes a signed 16-bit value on the HC16. The following example should help illustrate this.

Taking a register TOC2 (timer output compare 2 in the GPT module) with address \$FF916 as an example, we can read this with the following instructions:

1. LDD \$F916,X where XK = \$0, X = \$0000 signed offset
2. LDD - \$06EA,X where XK= \$0, X = \$0000 signed offset
3. LDD \$F916 where EK = \$F unsigned direct address
4. LDD \$16,X where XK = \$F, X = \$F900 unsigned offset

In number 1, \$F916 is a negative number and since a 16-bit offset is always signed, the XK register must be 0. Indeed, Number 2 is exactly the same instruction, but this time the negative number is shown more clearly by the negative sign. In number 3, the address mode has changed from indirect 16-bit signed offset on an index register to a direct address mode. In this case, the address is considered a positive, unsigned number, and so the EK register must be set to \$F to force the read from segment 15 (\$F). For completeness, number 4 shows the HC11 approach to this by using the unsigned 8-bit offset on an index register address. In this last example, the XK and X register form a 20-bit address and so the XK register must point to segment 15 (\$F) at the top of the address space.

For simplicity, the programmer would normally utilize number 3 to access the HC16 registers since this leaves the X index register for other activities within the program.

Making full use of the 16-bit offset addressing mode on the HC16 leads to other small changes in the user's software. Taking the previous example, we can look at it from the point of view of the HC11 user.

Normally, the HC11 TOC2 register would reside at address \$1018, and so the HC11 program would have:

- a. LDD \$18,X where X = \$1000
- b. LDD \$1018

Experienced users of the HC11 usually would opt for example (a) (or the equivalent using the Y register) because when using equates for the peripheral register address (for example LDD TOC2, where TOC2 = \$1018), it is important to remember that the bit manipulation (BRSET, BRCLR) instructions can use only 8-bit offsets. To avoid multiple definitions of the same register, most HC11 programmers keep the register address at a byte value and so can equally have LDD TOC2,X and BRSET #1, TOC2,X,LABEL (with TOC2 EQU \$18) to read the register. With the HC16, the EK register allows a 16-bit (2-byte) address to be used for the equate and so frees up the index register.

NOTE: *Bit manipulation now operates on both 8- and 16-bit addresses. Hence, for the HC16, with TOC2 EQU \$F916, the following would work perfectly.*

LDD	TOC2	(EK = \$F)
BRSET	#1,TOC2,LABEL	(EK = \$F)

Timing Changes

Program delay or timing loops are obviously affected by the change to the cycle times for instructions. There is no relation between the HC11 instruction timing and the CPU16 timing. Remember also that the HC16 can run at clock speeds of up to 16.78 MHz (bus speed of up to 8.39 MHz).

On the HC11, there are advantages in using the X index register rather than the Y index register. This is because the instructions using index Y require an additional opcode and so take one byte more memory and one more bus cycle to execute. The CPU16 does not have such a difference between the index registers which have the same number of opcodes and timing.

Assembly Code Differences Between HC11 and HC16

A number of HC11 instructions are changed slightly to allow them to function with the new core design. The list in [Table 4](#) is not a long one, and often the user can create a macro routine to instruct the assembler to change the HC11 code into CPU16 code.

The new instructions ANDP, AIX, AIY, AIS, PSHM, PULM, and ORP each replace several HC11 instructions. In particular, AIX #2 will often replace either two sequential INX instructions (similarly, AIX #-2 replaces two sequential DEX instructions) and so will make the CPU16 code smaller and look better.

The remaining changes are to accommodate the different stack operation of the CPU16 (discussed later in [Exception Handling \(Interrupts\)](#)) and the 20-bit addressing capability of the core.

Table 4. HC11 Instructions Modified for CPU16 Implementation

HC11 Instruction	Change in CPU16
BSR	Generates a different stack frame
CLC, CLI, CLV	Replaced by ANDP instruction
DES, DEX, DEY	Replaced by AIS, AIX, and AIY instructions
INS, INX, INY	Replaced by AIS, AIX, and AIY instructions
JMP	Indirect 8-bit offset replaced by 20-bit offset and extended addressing (20-bit address) modes
JSR	As JMP instruction and generates different stack frame
PSHX, PSHY	Replaced by PSHM
PULX, PULY	Replaced by PULM
RTI	Only unstacks the PC and CCR registers
RTS	Only unstacks the PC and PK registers
SEC, SEI, SEV	Replaced by ORP instruction
TAP, TPA	CCR bits and interrupt masking differ from HC11
TSX, TSY	Adds 2 to SK:SP before transfer to XK:IX or YK:IY
TXS, TYS	Subtracts 2 from XK:IX / YK:IY before transfer to SK:SP
TXY, TYX	Transfers full 20 bits, including K registers
WAI	Generates a different stack frame

Enhancements in CPU16 Source Code

20-Bit Addressing

The 16-bit signed branch (or long branch) limits the programmer to moving plus \$7FFF and minus \$8000 bytes from the current PC position. The HC11 JMP and JSP instructions have an operand of 16 bits and so allow a jump anywhere within the 68-Kbyte address range of a 16-bit operand. The JMP and JSR instructions have been changed on the CPU16 to take a 20-bit address. These instructions also allow a jump to an indexed IX, IY, or IZ address with a 20-bit signed offset. Thus, it is now possible to calculate the destination address of a jump with the HC16 where it would have required building and executing the instruction in RAM for the HC11. This instruction is not affected by the EK register which remains unchanged.

Moving Data without Affecting the Accumulators

Two new instructions have been added to CPU16 to move data from a specific memory location to another memory or range of memory locations. These are the MOV_B and MOV_W instructions which rather unsurprisingly are move byte and move word, respectively.

The instructions are in the form shown in [Table 5](#), with, of course, the same instructions for word moves.

Table 5. Move Instruction

MOV _B offset,X(n) EXT	Move byte from indexed address with post increment signed 8-bit offset to extended 16-bit address.
MOV _B EXT offset,X(n)	Move byte from extended 16-bit address to indexed address with post increment signed 8-bit offset.
MOV _B EXT EXT	Move byte from extended 16-bit address to extended 16-bit address.

This instruction is not completely orthogonal and does not include a move from indexed address to indexed address.

These instructions are particularly useful when capturing data from a peripheral module such as the SCI or ADC and placing the results in a buffer for future use. An interrupt routine could move the data with the minimum disturbance to the CPU registers and hence reduce the interrupt execution time. Such an example is shown here, where an interrupt routine triggered by the periodic interrupt timer (PIT) copies and A/D value to a circular buffer for an FIR filter.

Note also the use of the 32-bit load and store instructions LDED and STED.

Example 5. HC16 Moving Data from an ADC Result Register to a RAM Table

```
*****MOVEW A/D data into RAM *****
*           Interrupt routine running from PIT
*           128 word table for FIR filter
*
adresl1    equ    $F732
start_table equ    $0100
end_table  equ    $0200

RAMpoint   org    RAM
           rmb    4           ; 32-bit result

moveAD1    org    PIT_exception
           pshm   D,E,X,K     ; save altered CPU registers
           ldab  #$F         ; point to ADC
           lded  RAMpoint     ; get 32bit value containing 20bit pointer
           tbxk             ; put bits 19:16 into XK
           xgex             ; move bits 15:0 into IX
           movw  adresl1, x(2); get the data from ADC result register
           cpx  #end_table    ; check if IX pointing past end of RAM table
           bmi  OK           ; if not then continue
           ldx  #start_table  ; if yes, rest pointer to start of table
OK         xgex             ; get new value of IX into accE
           sted  RAMpoint     ; store new value in RAM again
           pulm  D,E,X,K     ; restore altered CPU registers
           rti
```

MAC and RMAC

The multiply-accumulate (MAC) instruction and the repeat MAC (RMAC) offer a substantial improvement in CPU performance over the HC11. As has been explained in the previous section about the core design, the MAC function has a number of dedicated registers in the CPU. These registers require the support of a number of new instructions, some of which can be seen in the example of code describing an FIR filter algorithm.

The new instructions are described here and provide the programmer with all the necessary tools to perform DSP algorithms.

First, the DSP section of the CPU must be initialized.

Table 6. DSP Register Initialization

LDHI	Loads the contents of the addresses pointed to by IX and IY into registers HR and IR, respectively
TDMSK	Loads the modulo addressing mask for the IX and IY registers.
CLRM	Zeros the M register and clears the appropriate flags in the CCR
TEM	Transfers the contents of acc E into bits 31:16 of the M register and clears the other bits in the M register
TEDM	Loads the M register with a 32-bit value in accE:accD

The MAC and RMAC instruction multiplies the HR and IR registers together and then adds the 32-bit result to the lower 32 bits of the M register.

The MAC and RMAC instructions are of the form:

MAC offsetX offsetY	(HR x IR) + M; IX and IY post incremented by offsets
RMAC offsetX offsetY	(HR x IR) + M; IX and IY post incremented by offsets Accumulator E decremented and then the calculation is repeated until acc E is 0

The flow of the MAC instruction is shown in **Figure 9**.

NOTE: All the CPU registers are modified by the MAC, with the exception of accumulator E which is used by the RMAC instruction to count the number of MAC instructions performed.

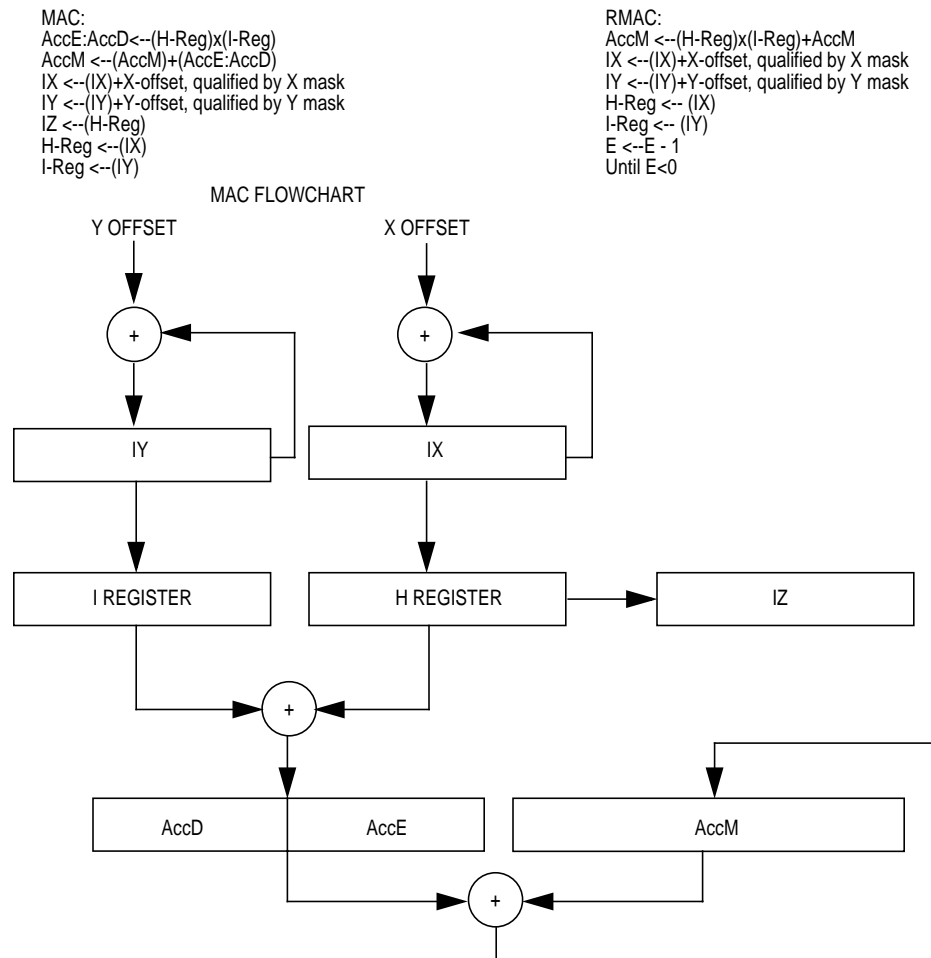


Figure 9. MAC Instruction Operation

The RMAC instruction is interruptable and takes 18 cycles for the first calculation followed by 12 cycles per iteration thereafter.

NOTE: The IX and IY modulo masks must be cleared to allow more than 256 indexed values to be multiplied by the RMAC instruction.

The four additional bits of the M accumulator allow a certain amount of overflow without the loss of data. The data in the M accumulator can be accessed by the commands listed in [Table 7](#).

Table 7. DSP Support Instructions

TMER	Transfer a convergent-rounded 16-bit value of M(31:0) into accE
TEMT	Transfer a truncated 16-bit value of accumulator M(31:16) into accE
TMXED	Transfer all 36 bits of accumulator M into IX(3:0):accE:accD with IX(15:4) sign extended with the value of bit 3 of IX
Support instructions also available:	
FMULS	Perform a signed fractional multiply of accE by accD and then shifts the result left one place and clear the accD(0) bit
ACED	Adds accE:accD to accumulator M(31:0)
ACE	Adds accE to accumulator M(31:16)
ASLM	Shifts the entire 36 bits of accumulator M left one place (multiply M by 2)
ASRM	Shifts the entire 36 bits of accumulator M right one place (divide M by 2)
PSHMAC	Save the contents of the HR, IR, M, and modulo masks on the stack
PULMAC	Restore the contents of the HR, IR, M, and modulo masks from the stack
LBEV	Long branch if the EV bit is set (M(31) set)
LBMV	Long branch if the MV bit is set (M(35) set)

Stack Operations

The differences between the cores mean that the stack operations will be quite different. The HC11 has a number of instructions for stack manipulation, but the stack is always made up of 8-bit values. The CPU16 stack consists of 16-bit values and has many more registers to save on the stack.

A fundamental difference is that only the CCR and PC are saved during an exception process. It is left to the programmer to decide which of the remaining CPU registers must be saved on the stack. To make life simpler, there are two new instructions to perform this task:

PSHM D,E,X,Y,Z,K	Saves any combination of listed registers to the stack in a fixed order
PSHMAC	Stacks all the MAC registers (M, HR, IR, X, Y masks)

PSHA and PSHB are retained from the HC11 source code to maintain compatibility, but PSHX and PSHY become PSHM X and PSHM Y, respectively, or can be performed with one line of code, PSHM X,Y, to save time.

The push multiple (PSHM) instruction adds the appropriate number of words to the stack and decrements the stack pointer appropriately.

It always makes sense to use as few CPU registers as possible when writing exception (interrupt) handlers for the HC16 as each additional stacked register takes two clock cycles to save it onto the stack and using the MAC registers (any of them requires 14 clock cycles for the stack operation). Remember also that they must all be pulled from the stack afterward, when the routine is complete and ready to return from interrupt.

The stack pointer is 20 bits on the CPU16 and can be placed anywhere in memory using the SK register. The SK and SP registers are either set up by including their address in the reset vector table (see [Initialization of HC16 Device](#)) or by the use of the TBSK and LDS instructions in the same way as the HC11.

NOTE: *Initialize the stack pointer at an even address (see [Example 10](#)).*

Table 8. HC11 Stack Control Instructions

BSR	Relative branch to subroutine (stack 2 bytes – PC reg)
DES	Decrement stack pointer by 1 byte
INS	Increment stack pointer by 1 byte
JSR	Direct jump to subroutine (stack 2 bytes – PC reg)
LDS	Load stack pointer with memory or immediate value
PSHA, PSHB	Push 8-bit A or B accumulators
PSHX, PSHY	Push 16-bit X or Y registers
PULA, PULB	Pull 8-bit A or B accumulators
PULX, PULY	Pull 16-bit X or Y registers
RTI	Return from interrupt (unstack 9 bytes)
RTS	Return from subroutine (unstack 2 bytes – PC register)
STS	Store stack pointer value in memory
SWI	Software interrupt (stack all 9 CPU registers)
TSX, TSY	Transfer stack pointer to X or Y register
TXS, TYS	Transfer X or Y register to stack pointer
Interrupt event	Stack 9 bytes (all CPU registers)

Difference between RTI and RTS

At first sight, the RTI and RTS instructions look very similar, but there is a subtle difference between them. They both pull the CCR and SP registers off the stack and both take 12 clock cycles to execute, but the RTI stores the entire CCR register contents while the RTS only restores the PK field of the CCR register.

Another small change is the fact that the RTI must return to the instruction following the one just completed prior to taking the exception. Since the pipeline fetches the code six cycles ahead, the RTI decrements the program counter by six and thus points to the correct place in memory after the exception.

Table 9. CPU16 Stack Control Instructions

BSR	Branch to subroutine (stack 4 bytes – PC, CCR)
CPS	Compare stack pointer with memory or immediate value
AIS	Add signed 16-bit value to stack pointer
JSR	Jump to subroutine (stack 4 bytes – PC, CCR)
LDS	Load stack pointer with memory or immediate value
PSHA, PSHB	Push 8-bit A or B accumulators
PSHM D, E, X, Y, Z, K, CCR	Push any combination of 16-bit registers in the list
PULA, PULB	Pull 8-bit A or B accumulators
PULM D, E, X, Y, Z, K, CCR	Pull any combination of 16-bit registers in the list
RTI	Return from interrupt (unstack 4 bytes – PC, CCR)
RTS	Return from subroutine (unstack 4 bytes – PC, CCR)
STS	Store stack pointer value in memory
SWI	Software interrupt (stack 4 bytes – PC, CCR)
TBSK	Transfer accB(3:0) to SK register
TSKB	Transfer SK register to accB and sign extend accB(7:4)
TSX, TSY, TSZ	Transfer stack pointer to X, Y, or Z register
TXS, TYS, TZS	Transfer X, Y, or Z register to stack pointer
Interrupt event	Stack 4 bytes (PC and CCR registers)

The RTS assumes that the jump or branch instruction was a 2-word instruction (except for a BSR label) and so just subtracts 2 from the PC after restoring the stacked PC and CCR. The BSR instruction adds 2 to the PC since it is a single word opcode, and so it simulates the JMP and LBSR opcode instructions of 2-word length.

By comparison, the HC11 takes 12 bus cycles for an RTI and five bus cycles for an RTS instruction, and the RTI pulls all the CPU registers off the stack.

16-Bit Signed Branch

To overcome the limitations of the HC11 branch instruction being limited to a signed 8-bit value, the CPU16 has a new set of instructions called long branches. These have a 16-bit signed offset and so allow a 32-Kbyte jump anywhere in program space, even across 64-Kbyte boundaries.

The instructions are simply the same as the 8-bit signed offset with the addition of the letter L before the instruction (for example, BRA REL8 becomes LBRA REL16). Since the 16-bit offset requires more data in the operand, the opcode becomes two bytes and the operand two bytes with a typical execution time of four to six cycles compared with the 8-bit relative branch which takes two or six cycles and has single-byte opcode and operand.

Pipelining

You will notice that there were two possible execution times for the conditional branch instructions. This is a function of the CPU16 architecture which involves pipelining the data read from memory and preprocessing the information before it is actually executed.

The pipeline is a 3-stage operation which first reads the 2-byte (word) value and then evaluates the opcodes. At this stage, the operands are evaluated and the instruction is executed. Finally, the opcodes are moved through to the third stage after execution is complete.

The improvements in CPU performance are approximately two-fold over the more conventional approach of the HC11 and hence the HC16 is roughly twice as fast as the HC11 for a given bus speed.

The pipeline is especially noticeable with two types of instructions.

The inherent instructions take two clock cycles (or one bus cycle in HC11 terms) to execute. Even the NOP instruction on the HC11 takes two bus cycles to execute.

The more important change, especially for the HC11 user, is that a conditional branch can take differing execution times depending upon whether the pipeline needs purging after the instruction is completed.

Let us take the BNE instruction. This takes three cycles for the HC11, irrespective of whether the condition is true or false. The CPU16 pipeline is set to read the next instruction as if the result is false and so it will read the next address after the BNE instruction while the BNE instruction is being evaluated and executed. If the result is false, then the CPU has already fetched and evaluated the next instruction and can immediately execute it, thus saving possibly four clock cycles. If, however, the BNE instruction result is true and the branch taken, then the CPU16 must look for the new address indicated by the relative offset in the operand and fetch this instruction instead. It will, therefore, need to disregard the first stages of the pipeline and start again. Hence, it takes six cycles this time to complete the BNE instruction.

The long branch is similar, but here the opcode plus operand now fill two of the three stages, and so only one stage of the pipeline must be disregarded if the branch is taken. Thus, the LBNE takes four cycles if not taken and six cycles if taken.

NOTE: *When converting code from HC11, take special note of the critical timing loops as it may be faster to invert the logic of the test to take advantage of the pipeline and speed up the execution.*

16- and 32-Bit Arithmetic

The following four examples show two pieces of code for the HC11 and then for the HC16 device. First, it will be striking how much shorter the HC16 code is compared to the HC11 code. Clearly, speed of execution is another important difference.

The first example comes from a linear interpolation table routine where a limited number of data points were used to form a complex table.

NOTE: *The HC16 code uses signed multiply and divide to allow the table to have positive and negative slopes.*

Example 8 and **Example 9** are cubing an 8-bit A/D result to give a non-linear function for an A/D input. Since the result is calculated each time and is always positive, unsigned arithmetic has been used in the HC16 example.

Example 6. HC11 Code to Calculate 16-Bit Times 8 Bits Divided by 16 Bits

```

CALC_TBL_ENTRY
    STD  NOMINATOR
    STX  DENOMINATOR
    LDAB MULTIPLIER
    MUL                      ; acc A x mult = R2
    STD  AMUL
    LDAA NOMINATOR+1
    LDAB MULTIPLIER
    MUL                      ; acc B x mult = R1
    STD  BMUL
    ADDA AMUL+1              ; added to first multiply gives low 16 bits
    STD  BMUL
    LDAB AMUL
    LDAA #0
    BCC  C_SKIP              ; carry not affected by ldaa or ldab
    INCB
C_SKIP
*
    LDX  DENOMINATOR
    LDD  AMUL+1              ; get top 16 bits of the 24-bit result
    TSTA                      ; check if multiply result was only 16 bits
    BEQ  C_SMALL              ; if yes then do a different divide
    FDIV                      ; calculation of low 8-bit result
    XGDX
    TAB
    CLRA
    RTS                      ; result in Acc D
*
C-SMALL
    LDD  BMUL
    IDIV                      ; calculation of high 8 bits (8-bit result)
    XGDX
    RTS                      ; result in Acc D

```

Example 7. HC16 Code to Perform 16-Bit Times 8-Bit Divided by 16 Bits

```

calc_tbl_entry
    emuls      ; multiplier x nominator
    edivs      ;-----> Acc D
                ; denominator
    xgdx      place result in accumulator D
    rts

```

Example 8. HC11 (8-Bit x 8-Bit x 8-Bit) / 9-Bit

```

*****
*      CALCULATE THE NUMBER OF PULSES FOR A, D, S, R
*      Acc A contains the A/D value for calculation
*      Acc D contains result
*
*      f(x) = (ad_measured + 15) cubed / 400
* or
*      f(x) = ( (AD_RESULT) x (AD_RESULT) x (AD_RESULT) ) / 400
*****
CALC_PULSES          ; save A/D value temporarily
                    STAA  TEMP_AD
                    TAB
                    MUL          ; A/D squared
                    CLR          CALC_SHIFT
READ_5              CMPA  #0
                    BEQ  READ_6  ; check if greater than 8-bit value
                    INC  CALC_SHIFT ; if yes then continue
                    LSRD          ; keep track of divisions
                    BRA  READ_5  ; divide by 2
READ_6              LDAA  TEMP_AD ; go back and check for 8-bit result
                    MUL          ; restore original A/D result
                    LDX  #400    ; and obtain cubed value
                    IDIV         ; offset for normalized values
                    XGDX         ; divide by offset for best values
READ_7              TST  CALC_SHIFT ; place 16-bit result in acc D
                    BEQ  READ_8  ; check if had previously div by 2
                    DEC  CALC_SHIFT ; if not then finish
                    LSLD         ; keep track of multiplies
                    BRA  READ_7  ; go back and check if all mults done
READ_8              RTS          ; result in Acc D

```

Example 9. HC16 (8-Bit x 8-Bit x 8-Bit) / 9-Bit

```

*****
*      calculate the number of pulses for a, d, s, r
*
*      acc B contains the A/D value for calculation
*      acc D contains result
*
*      f(x) = (ad_measured + 15) cubed / 400
* or
*      f(x) = ( (ad_result) x (ad_result) x (ad_result) ) / 400
*****
**
calc_pulses
    clra          ; ad_result
    tde          ; placed in accE (making sure upper byte=0)
    tba          ;
    mul          ; multiply accA and accB
    emul        ; Multiply accD and accE (=24 bits E:D)
    ldx         #400 ;
    ediv        ; divide E:D by 400 scale factor
    xgdx        ; place result in accD
    rts

```

Comparison of HC11 and HC16 Code and Benchmarks

A rough estimate of the increased performance of the HC16 over the HC11 is that at 16.78-MHz clock the HC16 is nine times faster at 8-bit operations than a 2-MHz HC11. Taking a theoretical 8-MHz bus speed HC11 and comparing it with the 16-MHz clock speed HC16 (8-MHz fast termination bus speed), the performance differential is still approximately a factor of 2. This is largely due to the improved opcode efficiency and pipeline of the CPU16 architecture.

For 16-bit and 32-bit calculations, the HC16 is even faster than the HC11.

Check List of Changes to HC11 Code

- Set the K registers up correctly.
- Ensure the CPU registers are stack in interrupt routines.
- Remember to add exception routines for the additional vectors.
- Initialize the SIM and peripherals MCR register at the start of the program.
- Change the register equate addresses if using similar peripheral functions.
- Check actual timing of software delay loops.
- Make use of the interrupt and arbitration priorities correctly.
- Remember that the HC16 averages nine times the speed of the 2-MHz bus HC11.
- Alter any code that manipulates the CCR. (The bits are moved about.)
- Check for misaligned stack addressing using PSHA, PSHB, etc.
- Indirect 16-bit offset address operations are now signed values.

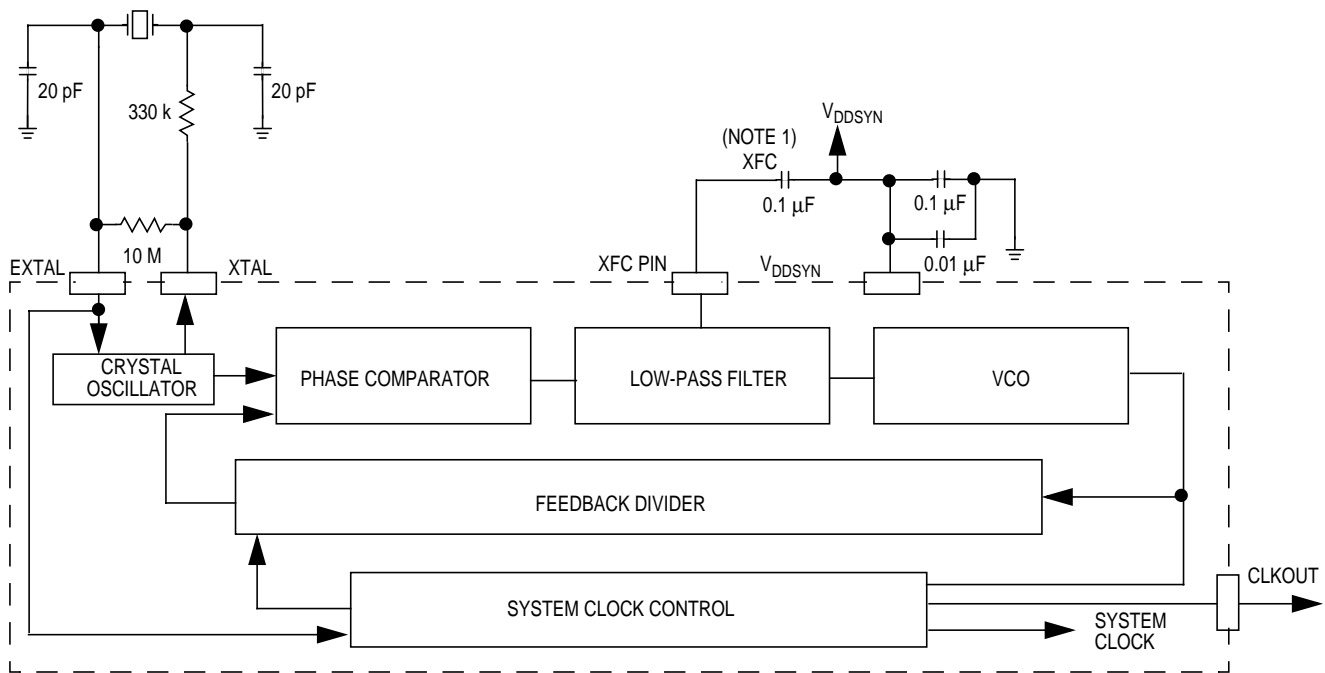
Initialization of HC16 Device

PLL Control

A significant change from the HC11 is the control of the clock frequency. The HC16 SIM module has a phase-locked loop (PLL) and a “limp mode” 8-bit oscillator built into the clock circuitry.

In normal operation, the SIM would have a 32-kHz crystal connected to the EXTAL and XTAL pins. This frequency is then multiplied and phase locked by the PLL to provide an internal clock frequency of up to 33 MHz in theory.

In practice, the maximum speed is currently restricted to 16.78 MHz.



Notes:

1. Must be low-leakage capacitor.
2. EXTAL can be driven with an external oscillator.

Figure 10. Schematic Diagram of the PLL

After reset, the PLL is preset at 8-MHz clock frequency from a 32-kHz crystal and so normally will be changed in the initialization section of the HC16 code.

NOTE: *There is a 20-ms lock time for the PLL and altering the W-bit will, therefore, take this amount of time before the frequency has stabilized. Changing the X bit has an instantaneous effect since it is outside the PLL feedback loop. As a consequence, it is necessary to check if a change to the X bit will take the oscillator over 16.78 MHz before the effect of a change to the W bit takes effect.*

An example is changing the oscillator from 8 MHz to 16.78 MHz where the W bit = 1 and X bit = 0 (SYNCR(15:8) = \$8F). Writing \$9F to the upper byte of SYNCR looks the same as writing \$7F to SYNCR(15:8), but the latter would cause the oscillator to attempt to run at 33 MHz for a short time as the PLL changes frequency and would certainly cause the system a fatal error. The correct procedure is to write \$3F to SYNCR(15:8) and then after the SLOCK bit is set to a 1 state (10 to 20 ms later) write \$9F to SYNCR(15:8).

A pin called MODCLK controls whether the PLL is enabled. Holding the MODCLK pin at a logic low during reset will disable the PLL and the system designer can then use an external high-frequency clock driver connected to the EXTAL pin.

NOTE: *It is not possible to connect a 16-MHz crystal to the EXTAL and XTAL pins directly.*

SRAM Initialization Procedure

The internal SRAM is disabled at reset. This is because the RAM is fully relocatable and the default start address is at the same location as the reset vectors. The SRAM base address registers should, therefore, be written and then the module configuration register written to enable the SRAM.

NOTE: *The SRAM base address registers cannot be modified unless bit 15 (STOP control) is set to a 1 (SRAM disabled) and bit 11 (RLCK: RAM base address lock) is cleared. The RLCK bit is a 1-time writable register after reset.*

When using the HC16Z1 evaluation board, it is often the case that a memory display window will be pointing to the RAM after it has been initialized. Typing the reset command will cause the HC16 device to be reset and the SRAM to be disabled again. The window will now show garbage data unless it is pointing to a chip-selected address. In addition, any further EVB16 commands will try to update the memory display window and could cause a string of DSACK errors as the EVB16 software reads non-existent memory.

Stack Initialization

Since the SRAM can be placed anywhere in the 1-Mbyte address space, it is obvious that the stack pointers will very likely be required to have the same flexibility. The 20-bit stack pointer is set up by the reset vectors, and so it seems a good idea to move the SRAM to this location as soon as possible after reset.

Remember that the stack will decrement by two bytes and so the stack pointer is always set to an even address. As an example, assume that the SRAM is positioned at \$14000 and since the MC68HC16Z1 has two Kbytes of SRAM, the top address is byte addressable RAM is \$143FF. The stack would be initialized with the value SK = \$1 and SP = \$43FE.

When debugging code with the HC16Z1 evaluation board, the user may find debug of code simpler by adding initialization of the stack pointer to the initialization code, even though it is loaded after issuing the RESET command on the EVB. Such an example is shown in [Example 10](#).

K Register Initialization

The 4-bit K registers are normally set up in the initialization routine of the HC16 along with any index register initialization just as one would do in HC11 code. Since the only way to access the K registers is via accumulator B, it is a simple matter to set up accB and copy the value into the appropriate K register.

The following code illustrates MC68HC16Z1 initialization of the SRAM, stack pointer, CSBOOT, GPT module and the K registers. By comparison, the HC11 routine is quite tiny.

Example 10. MC68HC16Z1 Initialization Routine

```

reset  ldab #1
       tbzk
       tbsk
       lds $ramstart+$03fe
       ldab #$0f
       tbek
       ldab #rambase      ; init rambase (=1)
       stab $fb05        ; rambah(low)
       ldd #ramstart     ; (=4000)
       std $fb06         ; lower 16 bits = $4000
       clr $fb00         ; rammcr = $8000
       ldd #$78b0
       std csorbt        ; set up csboot with 1 wait state
       ldab #$7f         ; 16.777 mhz
       stab $fa04        ; syncr
       clr $fa21         ; sypcr
       ldab #0           ; =0 for movedata subroutine
       tbxk
       ldab #1
       tbyk
       ldz #0000
       ldd #$000f        ; iarb=$f
       std gptmcr
       ldd #$9650        ; overflow=highest priority, irq level 6, vect=5x
       std gpticr
       andp #$ff1f       ; set cpu interrupt priority to 0 (lowest)

```

Example 11. HC11 Initialization Code

```

RESET  LDS    #$FF
       CLI
       ; CLEAR INTERRUPT INHIBIT

```

Table 10. Registers that Must/Should be Written after Reset

SIMMCR	SIM module configuration register
QSMPCR	QSM module configuration register
ADCMCR	ADC module configuration register
GPTMCR	GPT module configuration register
SRAMMCR	SRAM module configuration register
SYNCR	PLL control register
CSPAR0	Action of chip select pins
CSPAR1	Action of chip select pins
CSPDR	Chip-select port pins status – port C
PORTF	Port F output pin status
DDRF	Mode of operation of port F
PFPAR	Bus control pin status – port F

Table 11. List of 1-Time Write Bits/Registers

SIMMCR (MM bit)	SIM module address
SRAMMCR (RAMBAR bit)	SRAM base address
SYPCCR (entire register)	System protection control register

Mode Selection During Reset

The SIM can be configured in several different ways out of reset. This allows for the basic functions of asynchronous data/address bus, data/program space selection and chip selects. To bring the HC16 out of reset in a particular mode of operation, simply pull the appropriate data bus pin to logic 0 during reset. The best way to force the data pin is to connect the data bus to the outputs of a 74HC244 device and enable to outputs with the reset signal. Alternatively, a 2-k Ω resistor can be used to condition the data bus pins with a small effect on the switching characteristics of the data bus. The table of possible mode options is shown in [Table 12](#).

Reset Operation

Reset of the HC16 is much like the HC11. An address is fetched from a known area in the memory map and used to determine the start of the user's program.

The HC11 vector table contains the reset vector and interrupt vectors starting at \$FFFE,\$FFFF (16-bit address) and moving down the memory map two bytes at a time with increasing priority. See [Example 12](#) for an example of the HC11 vector table.

Table 12. SIM Configuration Out of Reset

Mode Select Pin	Default Function Pin Left High	Alternate Function Pin Pulled Low
DB0	\overline{CSBOOT} 16-bit	\overline{CSBOOT} 8-bit
DB1	$\overline{CS0}$ 16-bit $\overline{CS1}$ 16-bit $\overline{CS2}$ 16-bit	\overline{BR} \overline{BG} \overline{BGACK}
DB2	$\overline{CS3}$ 16-bit $\overline{CS4}$ 16-bit $\overline{CS5}$ 16-bit	FC0 FC1 FC2
DB3 DB4 DB5 DB6 DB7	$\overline{CS6}$ 16-bit $\overline{CS7-CS6}$ 16-bit $\overline{CS8-CS6}$ 16-bit $\overline{CS9-CS6}$ 16-bit $\overline{CS10-CS6}$ 16-bit	A19 A20–A19 A21–A19 A22–A19 A23–A19
DB8	Bus control $\overline{DSACK0}$, $\overline{DSACK1}$, \overline{AVEC} , \overline{DS} , \overline{AS} , \overline{SIZE}	PORTE
DB9	$\overline{IRQ7-IRQ1}$ MODCK	PORTF
DB11	Slave mode disabled	Slave mode enabled
MODCK	VCO = system clock	EXTAL = system clock
BKPT	Background mode disabled	Background mode enabled

The HC16 has a similar approach with the exception that the table starts at \$00000 and moves up the memory two bytes at a time. Prioritization of the interrupts is described in [Exception Handling \(Interrupts\)](#) and differs significantly from the HC11. The HC16 reset vector actually consists of four 16-bit values. These are the K register values for the ZK, SK, and PK registers, the 16-bit reset vector, the 16-bit stack pointer address, and lastly the 16-bit IZ register value. In effect, this is the 20-bit address of the reset vector and stack pointer plus a direct page address of 20 bits.

The reset vector can, therefore, jump anywhere in the 1-Mbyte address range of the program space, but all other vectors are just 16 bits and so can only jump directly to the first 64-Kbyte address segment of program space. From there, a JMP 20-bit address instruction will go anywhere in program space memory.

The reason for including the IZ register is for compatibility with the HC11 direct page instructions. These offer code efficient 8-bit operations and now have the added feature of being anywhere in memory, rather than just between addresses \$0000 and \$00FF in the HC11.

Vectors, Stack Operations

The first 512 bytes of address bank 0 (from \$00000 to \$001FF) are reserved for the vector or exception table. The first eight bytes are special, since these contain the reset vector data. This is followed by a number of special vectors required to maintain system operation. These are directly related to the SIM and CPU operation. After this is space for 200 user-defined vectors to be stored, each a 2-byte (16-bit) address pointing to address bank 0 (\$00200 to \$0FFFF).

There are many differences from the HC11 vector table. The most obvious is that the vector table starts at the beginning of the memory map and increments up through the map compared with the HC11 vector table starting at \$FFFE,F and decrementing the address two bytes at a time for each vector. The HC11 has fixed vectors always 16 bits in length, but the HC16 has vectors set up for most peripheral modules by the user and the reset vector has substantially more information in eight bytes. There are also a number of new vectors not familiar to the HC11 user. These are explained later.

Exception Routine Address

There are many more interrupt sources possible in an HC16 system. To allow maximum flexibility and maintain compatibility with the 68000 exception handling and external bus protocol, the internal peripherals do not have fixed predetermined exception vectors like the HC11.

Each peripheral may have the exception vector number initialized after reset, and it is up to the user to maintain these correctly. In addition, the scheme allows any external device to be initialized with its own vector number so there are no restrictions on the mix of peripheral devices.

User-Defined Vectors

The user vector number is initialized for each module or interrupt source. Some modules or submodules may have just a single vector, such as the PIT, while others, such as the GPT, have 12 vectors associated with the module. The vector number is an 8-bit value that, when multiplied by 2, becomes the vector address of the first exception vector for that module or peripheral.

An example of the HC16 vector table is shown in [Example 13](#). This shows that for the GPT the vector number is set to \$38 (decimal 56).

This can be compared with the definition of the HC11 and HC16 vector tables that follow. Note the different start addresses and direction down the vector table.

New Vectors for HC11 Users

The divide-by-zero interrupt is caused by a failure of the extended divide instructions from the CPU16. The bus error exception is explained in [External Hardware Interfacing](#), covering hardware design and basically occurs as a result of a read of non-existent memory. The breakpoint vector is associated with background mode. This leaves a spurious interrupt where the SIM cannot determine the source of the interrupt, the uninitialized interrupt, which is the default value for an interrupt on 68000 peripheral devices and, finally, the autovectors. The latter are most likely treated as XIRQ and IRQ pins by HC11 users. These are directly linked to the IRQ1 to IRQ7 pins of the SIM and provide the user with general-purpose interrupt vectors for non-68000 peripheral devices.

Remember that the order of the vectors is not linked to the priority of the interrupt and so, for example, the IRQ1 vector comes before the IRQ2 vector. There is, of course, a hierarchy of priorities for the exception processing, and this can be found in the SIM reference manual.

Reset Status Register

After a reset, this register can be checked to determine the source of the reset for the HC16 device. This register is at \$FFA06.

Sources of reset are:

- External reset
- Power-up reset
- Software watchdog reset
- Halt monitor reset
- Loss of clock reset
- System reset; from CPU32 and not available from CPU16

Example 12. Definition of HC11 Vector Table

```
VECTORS      ORG    $FFD6
              FDB    SCI_interrupt    ; SCI
              FDB    SPI_interrupt    ; SPI
              FDB    PAC_interrupt    ; PULSE ACC INPUT
              FDB    PAC_overflow     ; PULSE ACC OVERFLOW
              FDB    OVERFLOW         ; TIMER OVERFLOW 1
              FDB    IC4_INT          ; INPUT CAPTURE 4 / OUTPUT COMPARE 5
              FDB    OC4_INT          ; OUTPUT COMPARE 4
              FDB    OC3_INT          ; OUTPUT COMPARE 3
              FDB    OC2_INT          ; OUTPUT COMPARE 2
              FDB    OC1_INT          ; OUTPUT COMPARE 1
              FDB    IC3_INT          ; INPUT CAPTURE 3
              FDB    IC2_INT          ; INPUT CAPTURE 2
              FDB    IC1_INT          ; INPUT CAPTURE 1
              FDB    RTI              ; REAL TIME INTERRUPT
              FDB    IRQ              ; IRQ
              FDB    XIRQ             ; XIRQ
              FDB    swi_interrupt    ; SWI
              FDB    illegal          ; ILLEGAL OPCODE
              FDB    cop_interrupt    ; COP
              FDB    RESET            ; CLOCK MONITOR
              FDB    RESET            ; RESET
```


Table 13. Vector Table Definition for the HC16

Vector Number	Vector Address	SPACE (prog/data)	Exception
0	0000	P	Reset — Initialize ZK, SK, PK
1	0002	P	Reset — Initial PC (start of user prog)
2	0004	P	Reset — Initialize SP
3	0006	P	Reset — Initialize IZ (direct page)
4	0008	D	Breakpoint (BKPT)
5	000A	D	Bus error (BERR)
6	000C	D	Software interrupt (SWI)
7	000E	D	Illegal instruction
8	0010	D	Divide by zero
9-14	0012–001C	D	Unassigned
15	001E	D	Uninitialized interrupt
16	0020	D	Unassigned
17	0022	D	Level 1 interrupt vector (autovector)
18	0024	D	Level 2 interrupt vector (autovector)
19	0026	D	Level 3 interrupt vector (autovector)
20	0028	D	Level 4 interrupt vector (autovector)
21	002A	D	Level 5 interrupt vector (autovector)
22	002C	D	Level 6 interrupt vector (autovector)
23	002E	D	Level 7 interrupt vector (autovector)
24	0030	D	Spurious interrupt
25–55	0032–006E	D	Unassigned
56–255	0070–01FE	D	User-defined interrupt vectors

Example 13. Actual HC16 Vector Table

```

k          org    $0
sp        equ    $0110      ; not used, zk=$1, sk=$1, pk=$0
iz        equ    {ramstart+$03fe} ; stack pointer starts at address $43fe
          equ    $0000      ; index pointer set for registers

          org    vectors
          fdb    k          ; initial zk, sk, pk
          fdb    reset      ; initial program counter value
          fdb    sp        ; initial stack pointer value
          fdb    iz        ; initial direct page select (iz)
          fdb    bkpt_int   ; breakpoint address
          fdb    bus_err    ; bus error address
          fdb    swi_int    ; swi interrupt address
          fdb    illegal    ; illegal instruction address
          fdb    div_by_0   ; divide by zero
dumy1     rmb    6          ;
dumy2     rmb    6          ;
          fdb    un-initialised ; uninitialised interrupt
          fdb    reset      ; reserved
          fdb    int_1      ; level 1 interrupt autovector
          fdb    int_2      ; level 2 interrupt autovector
          fdb    int_3      ; level 3 interrupt autovector
          fdb    int_4      ; level 4 interrupt autovector
          fdb    int_5      ; level 5 interrupt autovector
          fdb    int_6      ; level 6 interrupt autovector
          fdb    int_7      ; level 7 interrupt autovector
          fdb    spurious   ; spurious interrupt

end_main_vect equ    *
          org    $70
          fdb    itic1      ; highest priority within GPT module
                               ; - any of below can be selected
          fdb    itic1
          fdb    itic2
          fdb    itic3
          fdb    itoc1
          fdb    oc2_int
          fdb    itoc3
          fdb    itoc4
          fdb    iti4o5
          fdb    ioverflow
          fdb    ipulse_ovr
          fdb    ipact1     ; lowest priority within GPT module

```

Exception Handling (Interrupts)

Interrupt Request Handling

Interrupts are called exceptions in the HC16 world. This conforms to the nomenclature of the 68000 processor and more accurately explains the nature of the sudden change in the activities of the CPU. Interrupts will, therefore, be referred to as exceptions from now on.

Module Design Influence on the Conversion from HC11 to HC16 Code

Many of the HC16 modules are clearly based on the functionality of an HC11 module. A prime example of this is the general-purpose timer (GPT) timer module. The main difference lies in the IMB interface and this required some specific initialization of the interrupt vectors, arbitration, etc. After this, there is often a modulus prescaler where the HC11 had a choice of just divide-by-two options before and this is necessary to accommodate the variable system clock frequency possible with the PLL. In addition to this, the module may also operate at higher maximum speeds, greater functionality, and generally be more flexible in its operation. All the peripheral modules have 16-bit wide registers and in general a much greater flexibility of operation than an equivalent HC11 module.

As an example, we will look at the GPT for the HC16 as we set up the module for interrupt handled events. This module is based upon the timer used on the HC11E9 and the HC11F1, but has a number of important differences.

Setting Up an Internal Exception

Both internal and external exceptions are basically handled the same way.

An exception starts when an interrupt source pulls an IMB or external IRQ line low (active state). The SIM responds with an interrupt cycle (CPU space) and waits for acknowledgment from the interrupt source. Any interrupt must generate an interrupt acknowledge (IACK) cycle. This lets the SIM know that the interrupt source is still available and it will then take control of the bus.

The SIM is able to determine that an internal module is an interrupt source due to the existence of a special line on the IMB called IIACK. This lets the SIM know that the interrupt source is still available and it will then take control of the address/data bus.

To set up an exception on an internal module, the interrupt level and arbitration priority must be set, along with the desired reset vector. Each module can and must be uniquely programmed with this data.

Remember that the exception vector must be doubled to provide the vector address since this device has a 16-bit vector address.

HC11 Timer Initialization

The task of initializing the HC11 timer is relatively simple. Set up the action of the event such as input capture or output compare, set up the output function pin state on the occurrence of the next event or the input trigger set up for an input capture, clear the event flag and enable the interrupt. **Example 14** shows the code for setting up OC1 and OC2 to cause a pulse to be generated from the OC2 pin.

Example 14. HC11 Code for Timer Initialization

```
*          TIMER_INITIALIZATION
*          set OC2 and OC1 for cylinder pulse on PA6
*          Interrupt on OC1 (falling edge of pulse)
*          timer set to 0.5 microsecond resolution
*
TIMER_SET
LDAA     #%01000000
STAA     OC1M+REGS      ; PA6 (OC2) is also controlled by OC1
LDAA     #%01000000
STAA     OC1D+REGS      ; OC1 set pin PA6 to 1 when compares
LDAA     #%01000000
STAA     TCTL1+REGS     ; OC2 toggles pin PA6 on compare
RTS
```

*HC16 Code
for GPT Interrupt
Initialization*

The HC16 has several different timers, but the GPT design is basically taken from the MC68HC11E9 timer. As a consequence, there is very little difference in the main part of the setup procedure. There are some additional lines of code to set up the GPT module exception vector base address, interrupt level, arbitration, and timer prescaler.

In the code in [Example 15](#), we can see the GPT being initialized for interrupts from output compare 2.

- NOTE:**
1. *The vector number (5) is part of the GPTICR register and sets up the vector number to be from \$50 onward. This means the vector table starts at \$A0.*
 2. *The GPTICR also contains the interrupt level (level 6 is used in this example) and the highest priority vector adjustment just as is possible with the HC11.*
 3. *The arbitration priority is set to the maximum of \$F. This is not required, but the timer is usually the highest priority event. But remember that the IRQ level is much more important than the arbitration level for prioritization of interrupts.*
 4. *Some registers only have half of the 16 bits written in this example. This is because the reset condition of the vectors is used.*
 5. *The output OC2 must be specifically made an output of the OC2 function. The default would be an input general-purpose port line.*
 6. *Just as with the HC11, the OC1 function can drive the OC2 pin.*
 7. *The interrupt capability is enabled finally and the timer is fully initialized. Remember to set the prescaler selection for the timer.*

Example 15. HC16 GPT Initialization for Interrupts on OC2

```

*
*GPT registers
*
gptmcr    equ    $f900
gptmtr    equ    $f902
gpticr    equ    $f904
gptport   equ    $f906
oc1       equ    $f908

:
toc1      equ    $f914
toc2      equ    $f916

:
tctl      equ    $f91e
tmsk     equ    $f920
*
                org    $A0
                fdb    ioverflow
                fdb    itic1
                fdb    itic2
                fdb    itic3
                fdb    itoc1
                fdb    oc2_int
                fdb    itoc3
                fdb    itoc4
                fdb    iti4o5
                fdb    ioverflow
                fdb    ipulse_ovr
                fdb    ipact1

*****
*           Timer_initialization
*
*           set oc2 and oc1 for cylinder pulse on oc2data
*           interrupt on oc1 (falling edge of pulse)
*           timer set to 0.5 microsecond resolution
*****

timer_set
        ldab    #$f
        tbek

        ldd    #$000f        ; iarb=$f
        std    gptmcr

        ldd    #$000f        ; overflow=highest priority
        std    gptmcr        ; irq level 6,vect=5x

        ldaa   %#00010000    ; oc2data (oc2) is also controlled by oc1
        ldab   %#00010000    ; oc1 set pin oc2data to 1 when compares
        std    oc1

        ldaa   %#00000001    ; oc2 toggles pin oc2data on compare
        staa  tctl

        ldd    #$1010        ; enable OC2 output
        std    gptport

        ldaa   %#00010000    ; oc2 output compare interrupt enable
        ldab   #$00000001    ; set timer prescaler to divide by 8
        staa  tmsk
        rts

```

Initializing the QSPI

The QSPI was briefly described previously. The QSM reference manual has a very good explanation of its use and good examples of initialization code. The code in [Example 16](#) initializes the QSPI. The four configuration registers are set up at the start of the user program and then normally only require changes in a few bits during the rest of the user software.

The steps are relatively simple:

1. Assign the pin functions in the QPAR register.
2. Configure the clock, SPI as master/slave, data size.
3. Set beginning and end of the queue, enable/disable wrap mode (continuous transmit) and enable interrupts.
4. Set up the command and transmit data in the appropriate queues.
5. Select any delay options as required and enable the QSPI.

The QSPI RAM model can be seen in [Figure 9](#).

Example 16. Initialization of the QSPI

```

QSPI_INIT
LDD    #$1B7E
STD    QPAR      ; PCS1 active, enable SPI pins, make MISO input
LDD    #$8008
STD    SPCR0     ; SPI master, inactive=0, leading edge, 1.05MHz
LDD    #$2069
STD    SPCR1     ; PCS to SCK delay=1.9µs, transfer delay=200µs
LDD    #$8F00
STD    SPCR2     ; SPI finished interrupt enable, end queue=$F
LDD    #$0200
STD    SPCR3     ; HALTA and MODE FAULT interrupt enabled

```

Setting Up an External Exception

The HC11 user is most likely to use the IRQx pins of the SIM to emulate the functions of the IRQ and XIRQ pins of the HC11. The HC16 requires that an autovector (AVEC) cycle be returned to the SIM to complete the interrupt acknowledge of an IRQ request. The simplest and cheapest way to do this is to use the built-in chip select control logic to generate an AVEC signal and so complete the interrupt acknowledge (IACK) cycle of the SIM. The later section on external memory control explains the chip select logic in more detail. There will be no need to use the chip select as an output since it will be purely for the generation of an internal signal for the SIM, but it does mean that particular chip select will be reserved for this purpose and cannot also be used as a chip-select function for external memory, etc.

In this instance, the chip select is a comparator checking for the CPU space identifier on the upper four bits of the address bus (A(23:20) which will be \$F. It also checks for the appropriate interrupt level that it is responding to by placing this data on address A(3:1). All the other address bits are set to logic 1. Hence, an autovector for level 2 will result in \$FFFF3 appearing on the address bus, and for autovector level 7 (IRQ7) the address bus will issue \$FFFFFF.

An example follows that illustrates the implementation of the chip-select logic to generate an AVEC signal.

This example also illustrates the initialization of the software watchdog. This operates in exactly the same way as the HC11. On starting the watchdog, it cannot be disabled until a reset occurs. A register called the reset status register (\$FFA06) contains a number of flags that show the last cause for reset in the HC16. One of these bits is a flag to indicate the existence of a watchdog reset.

Table 14. Generating an Autovector and Initializing the Watchdog

```
LDAB  #$C0      ; enable the watchdog (COP)
STAB  SYPCR     ; and set timeout period to 8 seconds

LDD   #$FFF8   ; initialize Chip Sel Base Reg for Autovector
STD   CSBAR3   ; on an IACK cycle: A24-A11=$FFF8, blK_sz = 2K
LDD   #$7801   ; initialize Chip Sel Option Reg for Autovector:
STD   CSOR3    ; asynchronous, any Interrupt Priority Level

LDAB  #$FF     ; set port F pins to be IRQ pins
STAB  PFPAR    ; this is redundant: it happens at reset

AUTOV:                ; when IRQ6 is low, this autovector routine starts
JSR   SEND_STRING
RTI                   ; return to the main loop
```


Periodic Interrupt vs. Real-Time Interrupt

The HC11 real-time interrupt (RTI) allows the user to generate “ticks” or regularly spaced events with four possible periods: 4.1 ms, 8.2 ms, 16.4 ms, and 32.8 ms with a 2-MHz HC11 device.

The HC16 has a much improved version of this called the periodic interval timer (PIT). As we have seen with other modules, the variable system clock frequency of the HC16 means that a modulus counter prescaler has replaced the four options of the HC11 to give a total of 255 options for timer periods, plus an additional divide by 512. This results in periods ranging from 122 microseconds up to 15.9 seconds.

The PIT clock source is the 32.78-kHz oscillator frequency (EXTAL) and not the system clock.

The PIT can be set to have an interrupt level from 1 to 7, and it is important to set up the vector number (multiplied by 2 to give the vector address).

NOTE: *No flag is set and hence there is no clearing mechanism other than the execution of the PIT exception handler after the interrupt occurs.*

The next example shows some example code of initialization of the PIT and also includes the vector and exception handler.

Table 15. Periodic Interval Timer Setup and Exception Handler

```

; PIT demo code.
picr      equ      $fa22
pitr      equ      $fa24

;          PIT INTERRUPT VECTOR
org       $70
fdb       pitint      ; PIT interrupt vector

;          PIT INITIALIZATION VECTOR
org       ROMCODE
ldd       #$0110      ; period = 1 second
std       pitr
ldd       #$0238      ; PIT vector $38, IRQ level 2
std       picr
andp     #$FF0F      ; set cpu interrupt mask = 0
bra      *

;          PIT INTERRUPT ROUTINE
;          assume EK=$F
pitint
        pshm      d,x
        ldx       #string
send_string
        ldaa      0,x
        beq       done
        ldab      $fc0c      ; scsr
        andb      #1          ; wait for TDRE=1
        beq       send_ch
        staa      $fc0f      ; scdr
        aix       #1
        bra       send_string
done     pulm     d,x
        rti

string   fcb      'The Z1 writes this using the PIT' , $0a, $0d, $00

```

Different Exception Levels

The CPU16 has seven discrete exception levels, compared with the HC11's two levels. Level 1 is the lowest exception mask level and level 7 is the equivalent of the non-maskable interrupt (NMI) of the HC11 (masked by its X bit in the CCR). Looking once again at the condition code registers in [Figure 6](#), it can be seen that there are bits I0, I1, and I2 on the CPU16 which replace the X and I bits of the HC11.

On the HC11, only the maskable interrupt (masked by the I bit) is available to the on-chip peripherals, whereas all seven levels are possible for on-chip peripherals on the HC16.

External interrupt pins IRQ1, IRQ2 up to IRQ7 can each generate an exception request if the feature is enabled.

In this case, IRQ7 is the direct equivalent to the XIRQ pin of the HC11, and any of the others can be used as the HC11 IRQ pin.

One fundamental difference is the fact that all internal and external exception requests are handled in exactly the same way by a single section of the system integration module (SIM).

Level 0 exception is effectively disabling the interrupt source.

Arbitration

The exception scheme is taken directly from the 68000 and later microprocessors where there could be very many different interrupt sources. Seven separate levels of exception mask is a great help, but in a complex system there is a need for an even greater control of exception priorities. For this reason, the concept of arbitration was developed and is implemented on the HC16 in the SIM to be precise. There are four arbitration bits available to the programmer that will allow 15 discrete levels of exception priority for each exception level. It is important that every exception source is given a non-zero arbitration priority so that only one exception source must have a unique arbitration priority for a given exception level or even the arbitration logic will be unable to differentiate between two sources of the same exception level.

Remember that even the external exception source must also have unique arbitration priorities. IARB = 1 is the lowest practical value of arbitration.

There is little chance of running out of exception levels and priorities as there are a total of 115 active levels.

An arbitration level of 0 means that the internal arbitration is switched off and so it will not be possible to have the CPU16 service an interrupt from such a source. This is because IARB = 0 disables the internal interrupt acknowledge (IIACK) signal and so the CPU cannot see the source of the interrupt and defaults to the spurious interrupt vector.

Same Exception Level

Several exceptions set at a single exception level are commonplace in HC11 applications and are equally possible on the HC16. Just as one would expect, an exception request of a given level locks out all other exceptions of that same level and lower. Similarly, as for the HC11, the exception vector is decided after the stacking operation of the CCR and PC registers is completed.

NOTE: *Here the CPU16 will stack fewer registers and consequently fetches the exception vectors very much faster than the HC11. The reduced exception latency is generally a great advantage when converting code from the HC11.*

Multiple Exception Events

When several exceptions occur simultaneously, the highest level of exception and arbitration will normally win. Also, a subsequent higher level of exception can interrupt a lower level exception routine.

For example, assuming a level 3 exception is in progress when a level 4 exception occurs. The level 4 exception is not masked out and will cause the system to stack the PC and CCR and then start executing the level 4 exception routine. The exception mask is now set to 4 and so a subsequent level 4 exception of any arbitration level cannot exception until the exception mask has dropped below level 4 once more. This will happen as the RTI of the first level 4 routine restores the CCR. At this point, the pending level 4 exception immediately takes control and the level 3 exception must wait a while longer before resuming its task. There is a short (2 μ s to 4 μ s) delay from the interrupt event occurring and the CPU fetching the vectors. During this time, other interrupts may occur of the same or higher interrupt level. It is during this period that the arbitration takes place.

See examples in [Figure 11](#) where three types of events can occur.

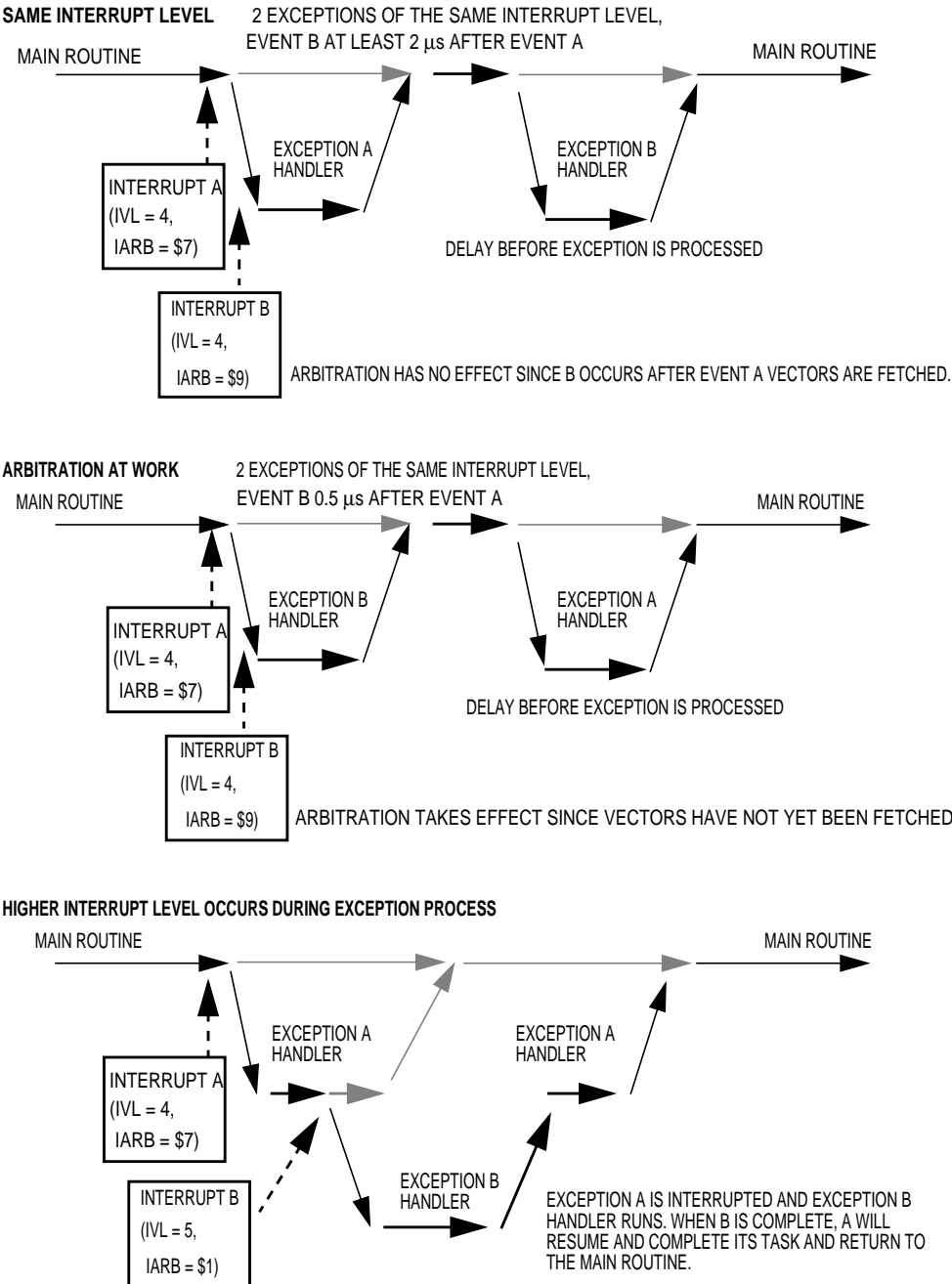


Figure 11. Multiple Interrupts

Prioritization Schemes

Prioritizing exceptions is normally self evident from the system specification, but when converting from HC11 code it is worth careful consideration where it is required on the HC16. The HC16 is much faster than the HC11 and most HC11 interrupt routines will not account for the possibility of being interrupted themselves, so leaving all routines at the same level of interrupt will normally work perfectly.

The flexibility of the exception scheme may seem rather daunting, but there are just a few key points to remember:

1. Simultaneous exceptions of the same interrupt level are arbitrated on the basis of the IARB0–IARB3 bits in each of the module control registers and so these must be unique.
2. An exception of the same level as the current exception mask cannot be executed until the mask level reduces below that pending exception level except for level 7 exceptions.
3. A higher level exception will interrupt a lower exception level routine which must then wait until the exception mask has returned to its level before continuing.
4. Level 7 exception is highest interrupt level and is a non-maskable exception that can be interrupted by another level 7 exception.
5. An arbitration level of 0 will cause a spurious interrupt if any interrupt occurs from that module.

Exception Routine Entry Latency

An HC11 interrupt takes the same amount of time as the SWI instruction to get into the interrupt routine (14 cycles) but the CPU16 exception must also clear the pipeline in addition to the operations common with the SWI command and so the SWI command takes 16 cycles while the exception takes 20 cycles (4 to reload 2 stages of the 3 stage pipeline).

If exception latencies are the reason for moving to the HC16, then remember that the first line of the exception routine is guaranteed to execute and could be a PSHM instruction. Making the first instruction a NOP will reduce the time from a maximum of 18 cycles down to just 2 cycles in addition to the 20 cycles entry latency and 38 cycles for the worst case instruction (EDIVS).

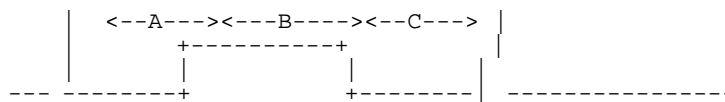
Included later are two exception routines that perform the same task, one for the MC68HC11E9 and the other for the MC68HC16A1. The similarity of the two routines is the most striking feature and emphasizes the ease of code conversion. The differences are easily seen and will invariably follow the same approach for any code conversion of an exception routine from HC11 to HC16.

The first difference is to remember to save the registers that are to be used in the exception routine, but not necessarily those that are used for global variables. As we can see, the accumulator D, IX, and K registers are all changed within the routine, but the IZ register is used as a global register and contains the base address of SRAM and so is not saved on the stack. At the end of the routine, the registers are pulled back off the stack with the same syntax as pushed onto the stack. The assembler orders the registers to ensure that the unstacking order is reversed.

The XK register is different in the exception routine and so must be initialized. This requires the use of accumulator B to make use of the TBXK instruction.

In terms of execution speed, the HC16 manages to perform the entire exception routine in 5.76 μ s where the HC11 takes 35.5 μ s. Also, the worst case latency reduces to 2.3 μ s from the HC11's 20.5 μ s.

Example 17. HC11 Timer Output Compare 2 Interrupt Routine



A + B + C = 331 cycles
A = 331 cycles - 258 ---> minimum low state (73 cycles)
B = PWM duty (256 - 2 cycles)
PWM frequency is 6.7 kHz (timer clock is same as bus clock)

Subroutine below takes 32 cycles (including interrupt latency) until the output compare 1 is re-armed, plus an IDIV or FDIV instruction may start executing with max 41 cycles left before done. Thus min period for this must be 331 cycles since the rising edge of the pwm must occur after the OC1 is re-armed.

Routine takes 71 bus cycles to complete (35.5 μ s at 2 MHz) with a worst case entry latency of 41 bus cycles (20.5 μ s at 2 MHz)

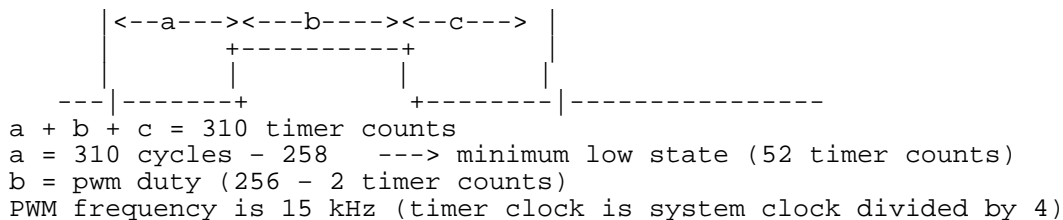
```
OC2_INT    LDX          TEMP_X           ; 4 interrupt latency
           LDD          TOC1+REGS       ; 5
           ADDD         #PERIOD          ; 4 256 (max PWM) + 2 + (max time) before
           STD          TOC1+REGS       ; 5 switch to other table is complete
           ; write OC1
```

```
* The OC1 reg has been updated so this is the point at which the routine
* must get to before OC1 has to go high. Worst case is when OC2 has
* maximum delay (max duty cycle).
           ADDD         2,X              ; 6 get new compare for OC2 (added to OC1)
           ADDD         #2              ; 4 prevent OC's occurring simultaneously
           STD          TOC2+REGS       ; 5 write OC2
           LDAA         #%01000000     ; 2
           STAA        TFLG1+REGS      ; 4 clear OC2 flag
```

* Note that OC2 only occurs soon after OC1 when there is plenty of time
* for the interrupt routine to execute (over 300 cycles) and so the
* OC2 flag may be cleared after updating the OC2 register.

```
           INX          ; 3
           INX          ; 3 increment X index by 2
           STX          TEMP_X         ; 4
           INC          TABLE_POINTER ; 6 increment A/D output counter
           RTI          ; 12
```

Example 18. HC16 Timer Output Compare 2 Exception Handler



The minimum pulse width is 2 timer clocks.

The PWM is 8 bits (256 timer clocks)

The subroutine below takes 52 cycles (including interrupt latency) until the output compare 1 is re-armed, plus an edivs instruction may start executing with max 38 cycles left before done. Thus the minimum period for this must be 90 cycles (23 timer counts) since the rising edge of the pwm must occur after the ocl is re-armed.

Entry conditions :

ZK:IZ points to SRAM

location temp_x,Z is temporary save of IX register

Routine takes 96 clock cycles (5.76 μ s at 16.66MHz

with a worst case entry latency of 38 clock cycles (2.28 μ s at 16.66MHz
; 20 interrupt latency

```

oc2_int  pshm      d,x,k      ; 10 save accD,IX and K on stack
         ldab      #1         ; 2  set up accB=1 ready for TBXK
         tbxk      ; 2       and point IX to data space
         ;         segment 1
         ldx       temp_x,z   ; 4
         ldd       toc1      ; 5
         add       #period    ; 4  256(max pwm) + 2 + (max time)
         ;         before switch to other table is complete
         std       toc1      ; 5  write ocl
  
```

* the ocl reg has been updated so this is the point at which the routine

* must get to before ocl has to go high. worst case is when oc2 has

* maximum delay (max duty cycle).

```

         add       2,x        ; 6 get new compare for oc2(added to ocl)
         add       #2        ; 4 prevent oc's occurring simultaneously
         std       toc2      ; 5 write oc2
         bclrw    tflg,oc2f  ; 4 clear oc2 flag
  
```

* note that oc2 only occurs soon after ocl when there is plenty of time

* for the exception routine to execute and so the

* oc2 flag may be cleared after updating the oc2 register

```

         aix       #2         ; 3 increment x index by 2
         stx       temp_x,z   ; 4
         inc       table_pointer,z ; 6 increment D/A output counter
         pulm     d,x,k      ;
         rti      ; 12
  
```


External Hardware Interfacing

Asynchronous vs. Synchronous Bus

The HC11 address and data bus is called a synchronous bus since all the timing is derived from the address strobe (AS) and E clock. The 68020 devices have an asynchronous bus where the MPU has a number of additional pins that act as handshakes for the bus communication. In effect, this allows a fast or slow external device to communicate with the MPU, and the MPU must wait for a handshake from the peripheral device before continuing with the portion of the bus cycle.

The HC16 uses the techniques from the 68020 devices but has the addition of chip-select circuitry to allow the user to control the entire bus cycle from the HC16. In the case of conversion from HC11 to HC16, it is almost certain that the user will make extensive use of the chip-select logic to reduce external hardware and access external peripheral devices.

The SIM reference manual describes the bus in great detail, but for a very basic look at the bus, we will look at using two external 6226 (128-Kbyte) SRAM devices with the HC16.

The chip-select logic lets the user program a specific set of conditions for the chip selects to function. This includes qualification with address/data strobes, address bus A(23:11), type of address space, etc., and the chip select has a base address and block size over which it will function.

To overcome the asynchronous bus, the chip-select logic generates the DSACK signals that would normally come from an external peripheral circuit. These can be programmed in terms of wait states. This refers to the number of bus cycles that the HC16 must wait before it can complete the bus cycle to read or write to the peripheral device. One wait state is the equivalent of 60 ns at a 16.66-MHz clock.

If the DSACK signal is not received within the time set up by the bus monitor (64 cycles default value), then the bus error (BERR) signal is asserted. This indicates the possibility of non-existent memory at this address. A second consecutive bus error indicates that the CPU is completely lost and so the SIM will force a reset.

NOTE: *The bus monitor is disabled after reset.*

The CSBOOT line is the only chip select to be active after reset. Its default settings are address strobe, upper and lower data bus read/writes, asynchronous mode, and 13 wait states. This allows for the slowest external memory and so will normally be altered immediately after reset to speed up the communication with the memory to the desired rate (normally, the fastest the memory can operate at). A chip select with zero wait states equates to a chip select access time of 85 ns (with a 16.78-MHz clock).

If there is only a single external memory, then the chip-select pins may not be required to be connected to the memory device. This would assume that the user had no other external bus devices and so no problems with bus contention. This would give an address access time of 114 ns at zero wait states, but remember that the chip-select logic must be configured for this memory or there will be no generation of DSACK when accessing external memory and the memory will not be addressable. The chip select logic is, therefore, essential to the operation of external memory even if the memory has the chip-enable pin permanently connected to ground.

Wait States

A wait state is a function of the asynchronous bus of the SIM. Normally, a 68000 system would have a mixture of different memory types with varying address access times. To accommodate this, the address/data bus sends out the address and then waits for handshaking signals before reading the data bus. To make this system operate as simply as possible, the chip-select logic in the SIM allows for a number of bus cycles to remain unchanged while the external memory fetches the data that it will output on the bus. By adding a clock cycle to the fastest normal bus cycle as a built-in delay is known as a wait state. At 16.66 MHz, this translates to 60 ns per wait state.

*Fast Termination
(Synchronous
Timing)*

Fast termination is a special case of address/data bus timing that is most commonly used with the internal peripheral modules and takes just two clock cycles to complete, compared with a zero wait state memory access of three clock cycles. In effect, this is the equivalent to the HC11 type of address/data bus cycle as it is synchronous with the CPU internal clock.

NOTE: *All the cycle times for instructions are shown in terms of fast termination. An external memory must have an access time of 35 ns to be able to operate fast termination mode with a 16.78 MHz HC16. This compares with the 4-MHz HC11F1 (non-multiplexed) memory address access time of 100 ns.*

Using Chip Selects

The chip selects take care of all the address and bus control compare logic required to drive external memory devices. A single-chip select can be eight or 16 bits wide and general chip-select signals and a range of addresses with a resolution of two Kbytes anywhere in the address range of the HC16 device.

Further options are upper/lower data bus, read, and/or write, wait states, type of address space, qualification on a read cycle with address strobe or data strobe and asynchronous or synchronous control.

The additional control of interrupt priority level (IPL) and AVEC are just for an alternative function of the chip-select logic to generate AVEC for an external IRQ as described in the previous section on exception processing.

*8-Bit and 16-Bit
Read/Write
Access to 8-Bit
Wide Memory
Devices*

In the following example, three chip selects are connected to two 32-k by 8 bits wide RAM devices (MCM60L256AP10) with 100 ns access time. CS2 is connected to the output enable pins of both devices, while CS0 and SC1 are connected to the write enable pins of the even addressed and odd addressed memory devices, respectively. This is the arrangement to be found for controlling the two optional RAMs (U1 and U3) on the HC16Z1 EVB.

We will put the memory at address range \$80000 up to \$8FFFF (64-k range). There are two 8-bit wide memories, and these attach in parallel to the 16-bit wide data bus. Address A0 effectively determines which device is written to, but A0 is not connected to the device because we will set up the chip-select logic to perform this task for us. **Figure 15** shows the actual device connections to the chip-select logic and HC16 bus. This all means that the base address for both memories is \$80000.

NOTE: *Although the memories are 32 Kbytes, the block size is 64 Kbytes with the two memories attached to different halves of the data bus.*

Since CS0 is connected to the RAM on the upper eight bits of the data bus, it is the even addresses, and CS1 controls writes to the RAM on the lower eight bits and becomes the odd addresses.

The memory devices in use are 100-ns access time, and since we are using the chip selects to enable the device, we must take the access times from the chip-select falling edge rather than the address bus valid time which is earlier. See **Appendix C** for a simplified timing diagram. This gives an access time required for zero wait states of 85 ns. This is clearly less time that the 100-ns device can manage, and so we must use one wait state timing which gives an extra 60-ns access time of 145 ns.

Example 19. Initialization Code for 8- and 16-Bit Addressing of External Memories

```

CSOR0      EQU          $FF4C
CSBAR0     EQU          $FF4E
CSOR1      EQU          $FF50
CSBAR1     EQU          $FF52
CSOR2      EQU          $FF54
CSBAR2     EQU          $FF56
CSPAR0     EQU          $FA44

LDAB       #$0F
TBEK
LDD        #$0803
STD        CSBAR0      ; set U1 RAM base addr to $80000: bank 3, 64k
STD        CSBAR1      ; set U3 RAM base addr to $8000: bank 3, 64k
LDD        #$50B0
STD        CSOR0       ; set Chip Select 0, upper byte, write only
LDD        #$30B0
STD        CSOR1       ; set Chip Select 1, lower byte, write only
LDD        #$0803
STD        CSBAR2      ; set Chip Select 2 to fire at base addr $80000
LDD        #$78B0
STD        CSOR2       ; set Chip Selects 2, both bytes, read and write
LDD        #$3FFF
STD        CSPAR0      ; set Chip Selects 0,1,2, to 16-bit ports

```

*Hardware for 8-Bit
and 16-Bit
Addressing Using a
Single-Chip Select*

It is normal practice to use the CSBOOT line to select two separate memory devices (often EPROMs) since this is the only chip select active following reset. In some cases, however, it may be desired to write both 8-bit and 16-bit values to a RAM which is selected using CSBOOT. An example of this is the HC16Z1 evaluation board where the main emulation memory is selected by CSBOOT. The HC16 can read an 8-bit value correctly with a single chip select connected to both memories because the CPU16 will only read the relevant data. Writing a byte is a different matter and in the same setup as earlier, an 8-bit write would cause a write to both memories with the same data in each byte. Thus, a write of \$55 would appear to be a write of \$5555.

To cure this, the chip select must be gated with signals that indicate a byte write and the address to be written. The chip select is set up for 16 bits, and rather than go directly to the output enable pins of both memory devices, is gated by the SIZ0 and A0 lines. The SIZ0 line is a logic 1 during a byte read or write operation and the address A0 line is used to determine which of the two memory devices is to be written to. SIZ1 indicates an even address read or write and so is not required in the equation that follows because the function of the extra logic is to prevent writes to the wrong memory device. This approach makes the external logic much simpler.

Thus a simple logic equation results in two chip-select lines with slightly delayed timing compared with the original chip select. The chip select delay is a maximum of 30 ns with the circuit described in [Figure 12](#). Thus, an 85-ns access time RAM would require an extra wait state to be added to the chip-select access time (zero wait states in 85 ns at 16.6 MHz).

Taking the chip select to be CSBOOT, we have the following equations for the high and low memory chip enable pins.

The logic equations are:

$$CSLOW = SIZ0 \cdot A0 + CSBOOT$$

$$CSHIGH = SIZ0 \cdot /A0 + CSBOOT$$

This can be done either by a PAL, such as a 16L8, or with two 74HC00 devices.

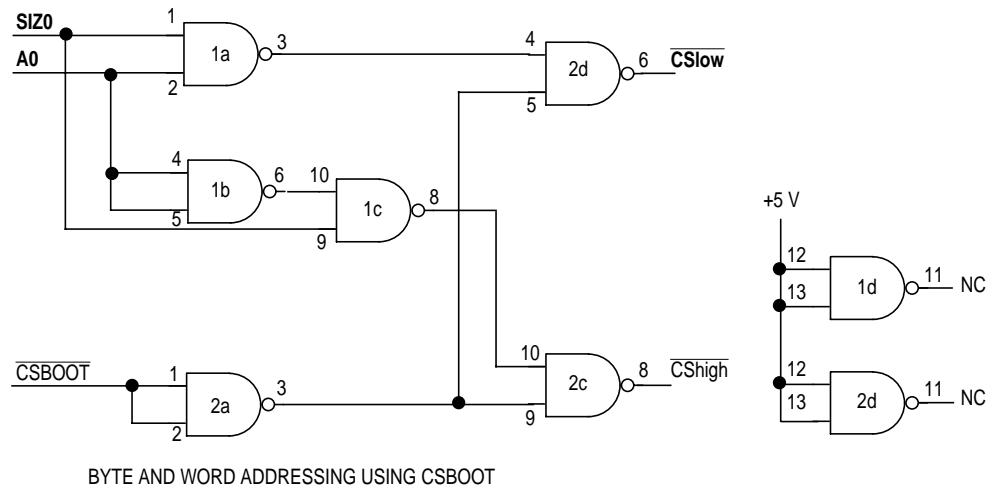


Figure 12. 8- and 16-Bit Address Read and Write Access with CSBOOT

**V_{DDE} vs. V_{DDI}
vs. V_{DDA}**

A device with so many different modules on board can be a real problem for the designers. Some modules generate a great deal of noise on the V_{DD} and V_{SS} supplies because they are switching loads very quickly (for example, address and data bus and I/O ports), while other modules require as noise free a supply as possible (for instance, ADC and PLL).

To overcome some of these problems, the HC16 has four separate power supplies.

V_{DDI} and V_{SSI} are the internal power supply pins. This powers up the CPU, SIM internal logic and the internal logic of other modules. Power consumption is relatively consistent, with few sudden changes in impedance.

V_{DDE} and V_{SSE} are the power supply for the external pin logic. This supply can have large swings in current consumption as I/O pins switch on and off.

V_{DDA} and V_{SSA} are the supply for the analog-to-digital converter (ADC). Since the ADC module has a 10-bit accuracy, it is important to have as little noise on these power supply pins as possible. A 10-bit accuracy means a resolution of just 3-mV peak to peak with a 3-volt reference voltage differential.

V_{DDSYN} is a special power supply just for the PLL circuit. Just as with the ADC, the PLL internal voltages are very susceptible to noise and the internal analog section of the circuit is affected by changes in voltage of as little as 5 mV. The effect of noise on V_{DDSYN} is to alter the VCO frequency by a small amount.

V_{STBY} for the SRAM standby power should be treated as for the V_{DDI} supply. It may also be grounded to avoid the SRAM being disabled by a lowering of the V_{DDI} supply below the V_{STBY} voltage.

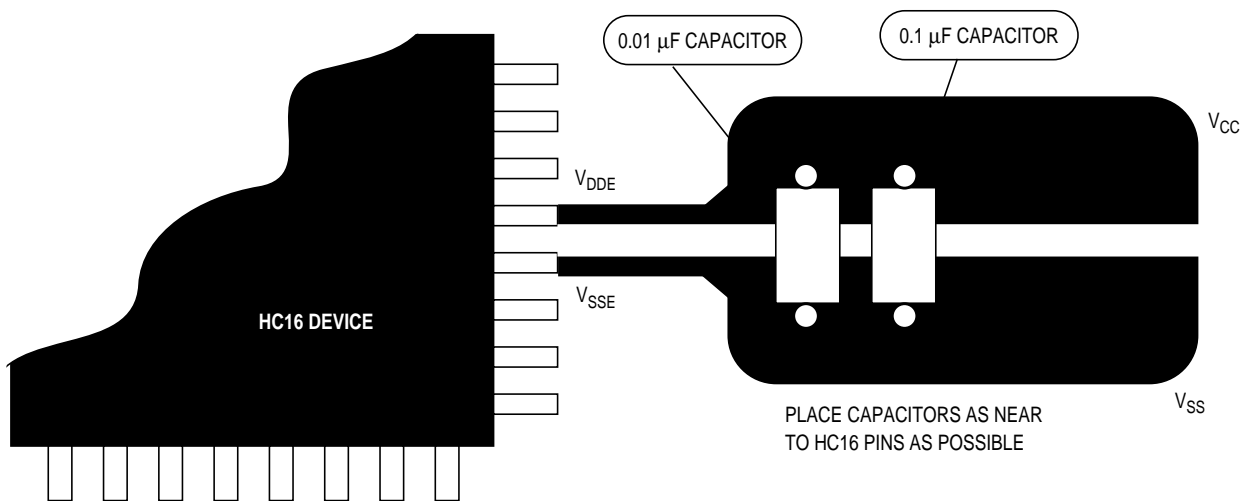


Figure 13. Shows the Suggested Decoupling as Close to the HC16 Pins as Possible for These Pairs of Power Pins

Minimum Required Connections for the SIM

The SIM is rather more complex than the address/data bus section of the HC11 and may take a while to learn, but many of the features have defaults that enable the user to place the device into an application with the minimum of fuss and with few external connections.

Figure 15 shows the HC16 in the same application as the HC11 in **Figure 14**.

NOTE: *Unused input pins should be tied to V_{SS} or V_{DD} via 10-k Ω resistors. Note the changes to the address and data bus interface and the connections to the SIM module in particular.*

Points to note about the HC16 application include:

1. The HC16 data bus has internal pullup resistors internally in the SIM.
2. There are several power supply pin pairs, all of which must be connected to power and each pair should be suppressed with a 0.01 μF and 0.1 μF capacitor to reduce system noise. Take particular care over noise suppression on V_{DDSYN} .
3. Mode selection after reset is set by the 74HC244 device.
4. The memories need doubling up to make full use of the speed of the HC16. A single 8-bit wide data bus can be used but reduces the performance of the system significantly.
5. Four chip selects are used. The EPROM is byte and word read access, and the RAM is byte and word read and write access.
6. The EPROM uses A15 to select the device on the lower 32-k part of a 64-k block, the CSBOOT line selects the bank (in this case bank0).
7. Many pins can be input or output depending upon the mode after reset, so pullup resistors have been shown in these cases. Many could be removed for specific modes of operation.

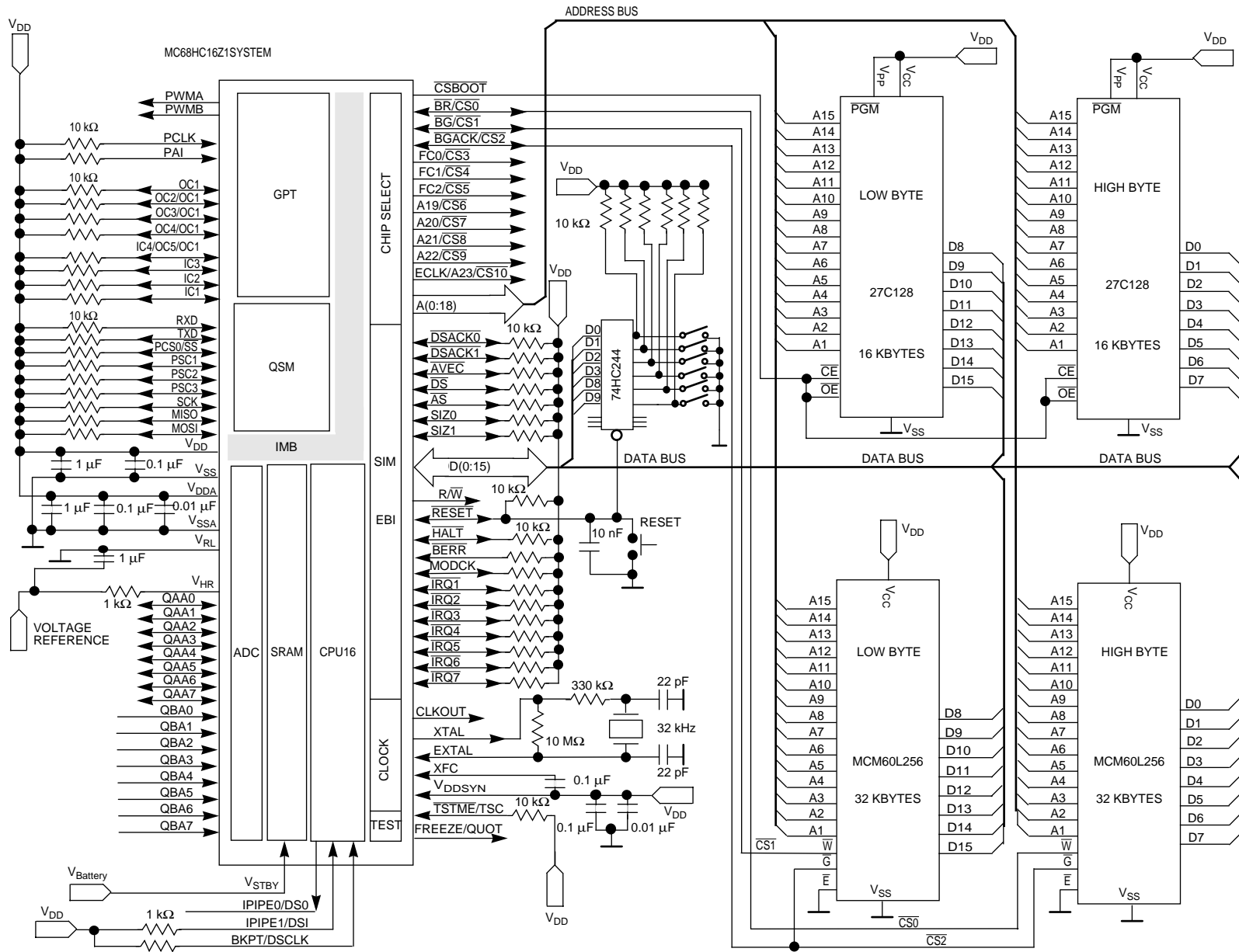


Figure 15. HC16 with External Memory

Debugging Tools

Background Mode Those experienced in using PCBUG11 for the HC11 device will find background mode on the HC16 very familiar. This mode is intended just for debug purposes and would not normally be used in a finished application, although the application could leave access to the mode for later problem solving.

Background debug mode uses three dedicated pins that are the only direct connection to the CPU16. The communication with the CPU16 is similar to the SPI synchronous serial communication, with a clock pin (DSCLK) and transmit and receive pins (DSO and DSI, respectively).

In general, the user will be unaware of the background mode operation, but it is used in most of the emulation systems and is the basis of the EVB for the MC68HC16. There is a small number of commands which are built up by the software running on a PC to make up the normal emulator commands. Thus, a dump of a block of 256 bytes of memory will be translated into 256 reads of memory contents at specified addresses which are sequential. The programmer can enter the background debug mode with the CPU16 instruction BGND.

An ample explanation of the background mode can be found in the *M68HC16 Family CPU Reference Manual*, Motorola document order number CPU16RM/AD.

Evaluation Board The EVB16 software on the IBM PC provides a sophisticated debug tool with very little external hardware. The EVB16Z1 emulator for the HC16Z1 device has sockets for 128 Kbytes of RAM, the HC16Z1 device, of course, and a background mode connector to the PC via the parallel port of the PC. There is little requirement for much else as the background debug mode and the PC software provide a high degree of debug capability, including 256 instructions trace buffer and assembly source level debug to name just two features. HP logic analyzer clips can be used to perform real-time analysis of code running on the EVB via the special groups of pins for these clips.

There is ample information available from Motorola concerning the EVB and other related products.

Appendix A

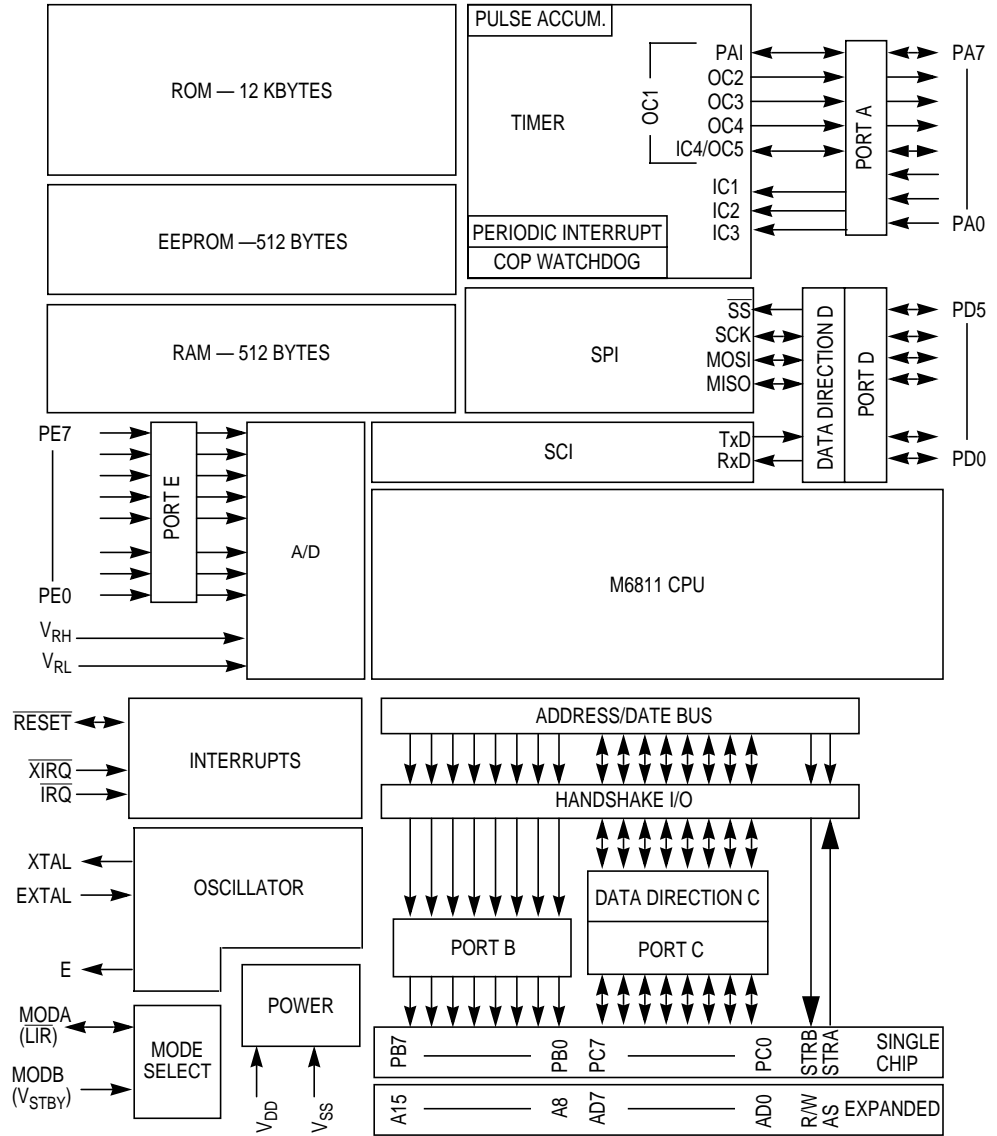


Figure 16. MC68HC11E9 Device

Appendix B

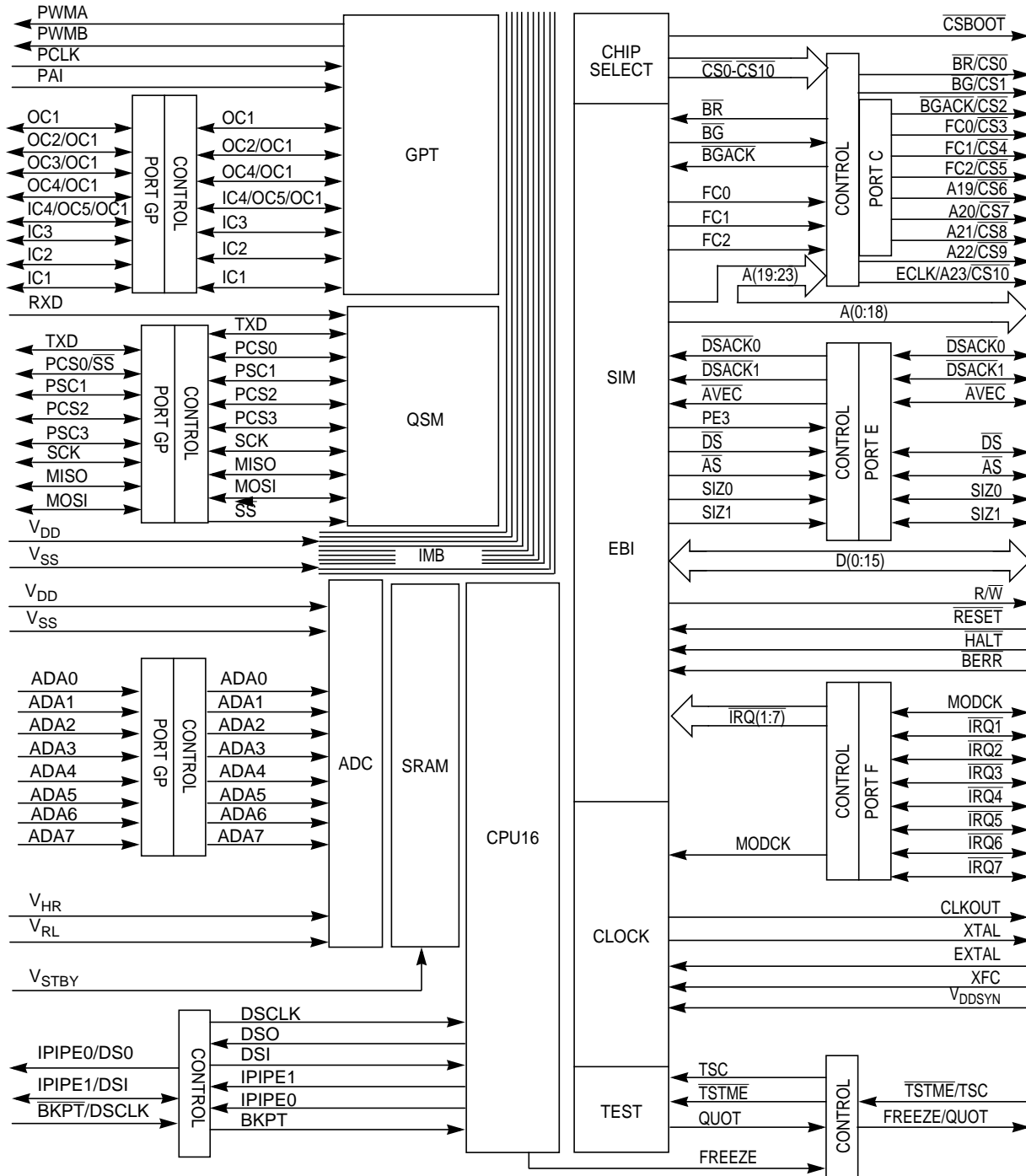
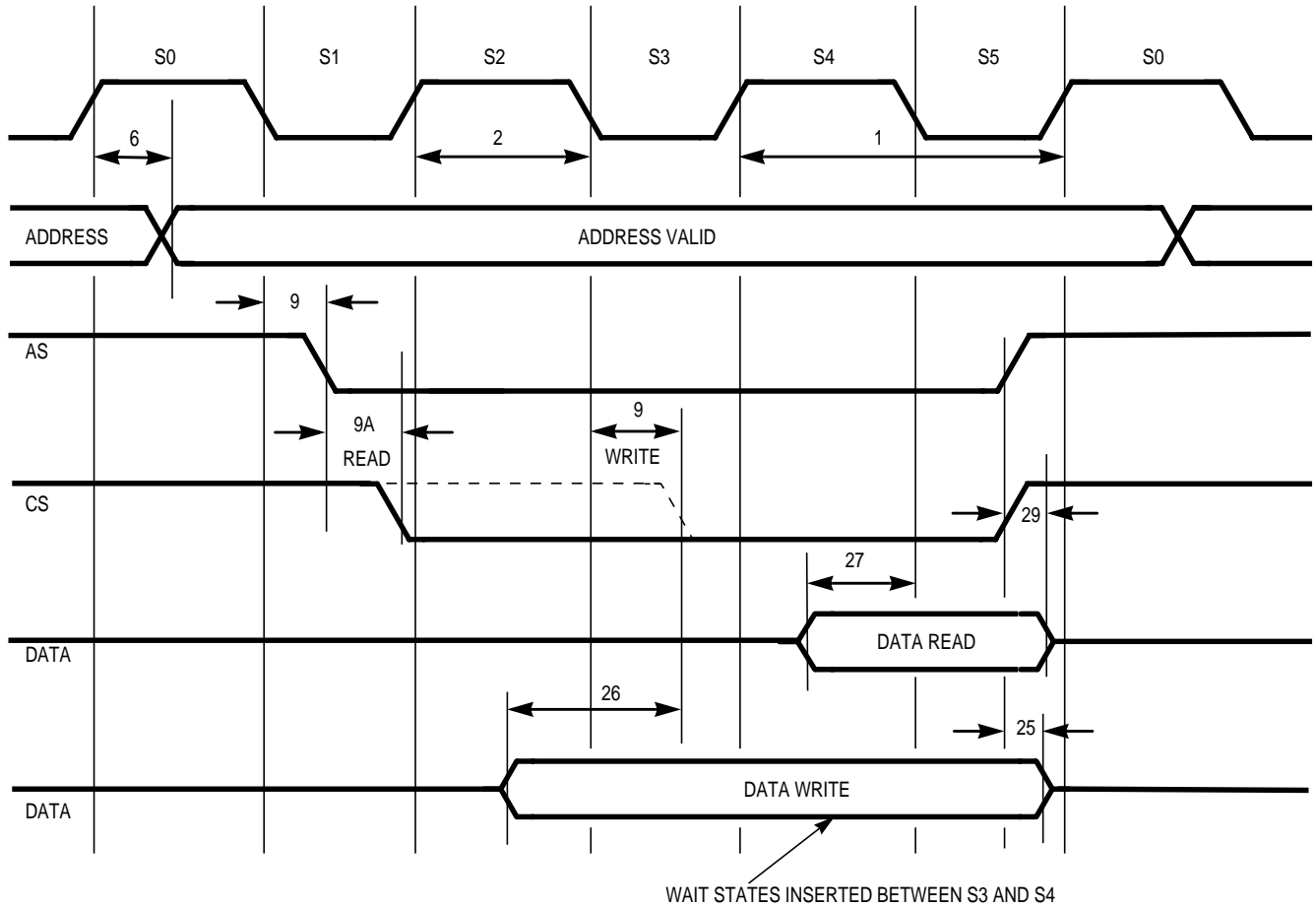


Figure 17. MC68HC16Z1 Device

Appendix C



TIMING PARAMETERS


1 - CLOCK PERIOD	MIN	60 ns
2 - HALF CLOCK PERIOD	MIN	28 ns
6 - CLOCK HIGH TO ADDRESS VALID	MAX	29 ns
9 - CLOCK LOW TO AS, DS, CS ASSERTED	MAX	25 ns
9A - AS TO DS OR CS ASSERTED (READ)	MAX	15 ns
25 - DS, CS NEGATED TO DATA OUT INVALID (DATA HOLD)	MIN	15 ns
26 - DATA OUT TO DS, CS ASSERTED (WRITE)	MIN	15 ns
27 - DATA IN TO CLOCK LOW (DATA SETUP)	MIN	5 ns
29 - DS, CS NEGATED TO DATA IN INVALID (DATA IN HOLD)	MIN	0 ns

AT 16.66 MHZ

CHIP SELECT	=	1	+ 1	- 9	- 27		
	=	60 ns	+ 60 ns	- 25 ns	- 5 ns	= 85 ns	
ADDRESS ACCESS	=	1	+ 1	+ 2	- 6	- 27	
	=	60 ns	+ 60 ns	+ 28 ns	- 29 ns	- 5 ns	= 114 ns

Figure 18. Simplified HC16 Timing Diagram

Application Note

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Motorola data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola and  are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

How to reach us:

USA/EUROPE/Locations Not Listed: Motorola Literature Distribution, P.O. Box 5405, Denver, Colorado 80217, 1-303-675-2140
or 1-800-441-2447. Customer Focus Center, 1-800-521-6274

JAPAN: Motorola Japan Ltd.; SPS, Technical Information Center, 3-20-1, Minami-Azabu, Minato-ku, Tokyo, 106-8573 Japan.
81-3-3440-8573

ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd.; Silicon Harbour Centre, 2 Dai King Street, Tai Po Industrial Estate,
Tai Po, N.T., Hong Kong. 852-26668334

Mfax™, Motorola Fax Back System: RMFAX0@email.sps.mot.com; <http://sps.motorola.com/mfax/>;
TOUCHTONE, 1-602-244-6609; US and Canada ONLY, 1-800-774-1848

HOME PAGE: <http://motorola.com/sps/>

Mfax is a trademark of Motorola, Inc.



MOTOROLA

© Motorola, Inc., 1992, 2000

AN461/D