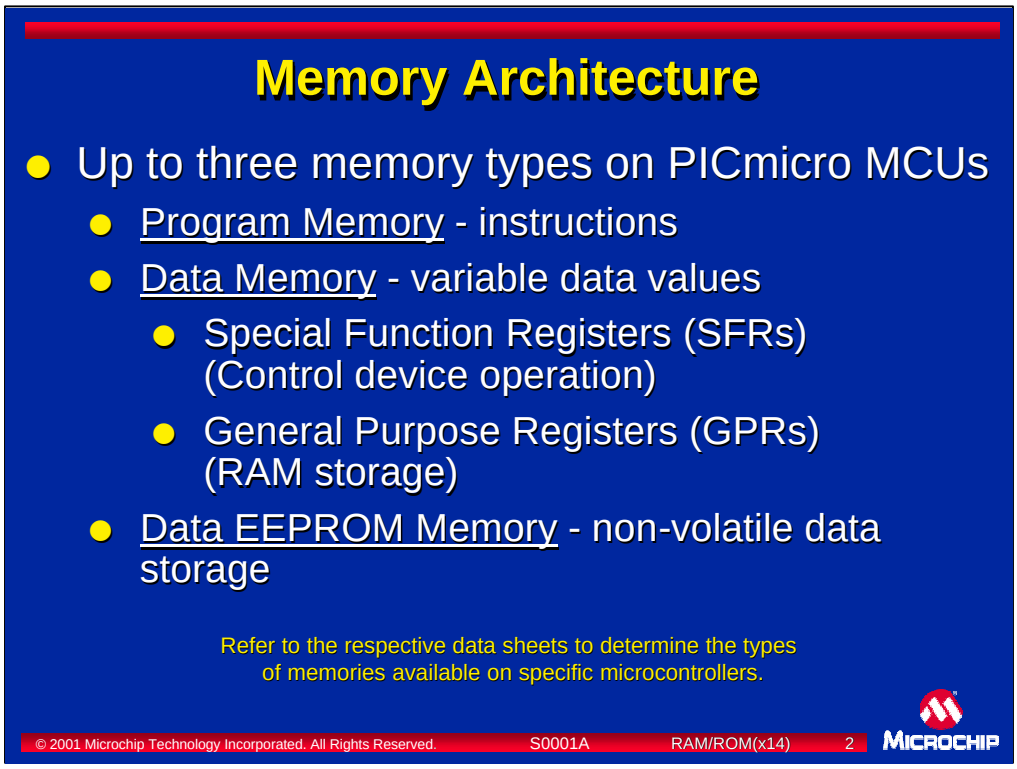
In this “Getting Started” tutorial you will learn about the various memory types found on Microchip’s PICmicro® microcontrollers (MCUs). This module will explain the three different memory spaces that can be used on the PICmicro devices and the type of memory used for each. The architecture of each memory space will also be explained, including addressing modes and code examples.

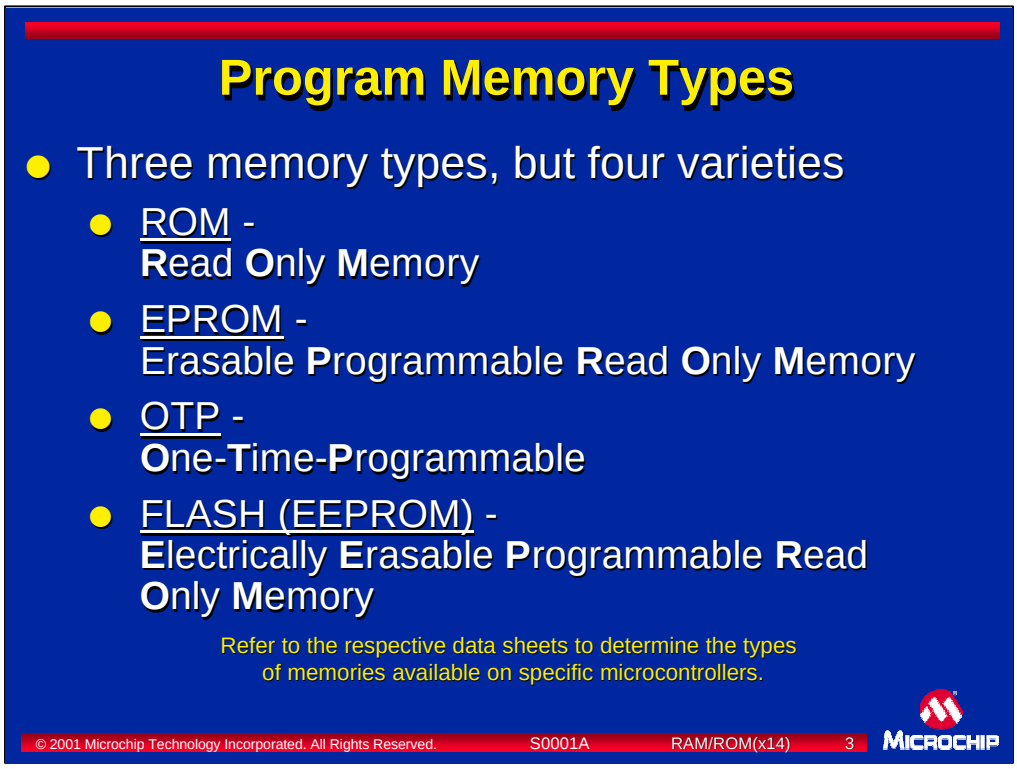


Each microcontroller may have up to three different memory types depending on the device family it belongs to.

The first memory type is common to ALL microcontrollers. This is called Program Memory, and its purpose is to store instructions.

The second memory type, Data Memory, is also common to ALL microcnontrollers. Data Memory contains the Special Function Registers and General Purpose RAM. The Special Function registers are commonly referred to as SFRs and consist of the configuration, control and status registers for peripheral and port I/O. General Purpose RAM, commonly referred to as GPR, is the area that the application firmware uses to store and manipulate data.

In addition to Program Memory and Data Memory, some microcontrollers also contain the third memory type, Data EEPROM Memory. The Data EEPROM Memory area of a microcontroller provides non-volatile data memory storage that can be rewritten many times.

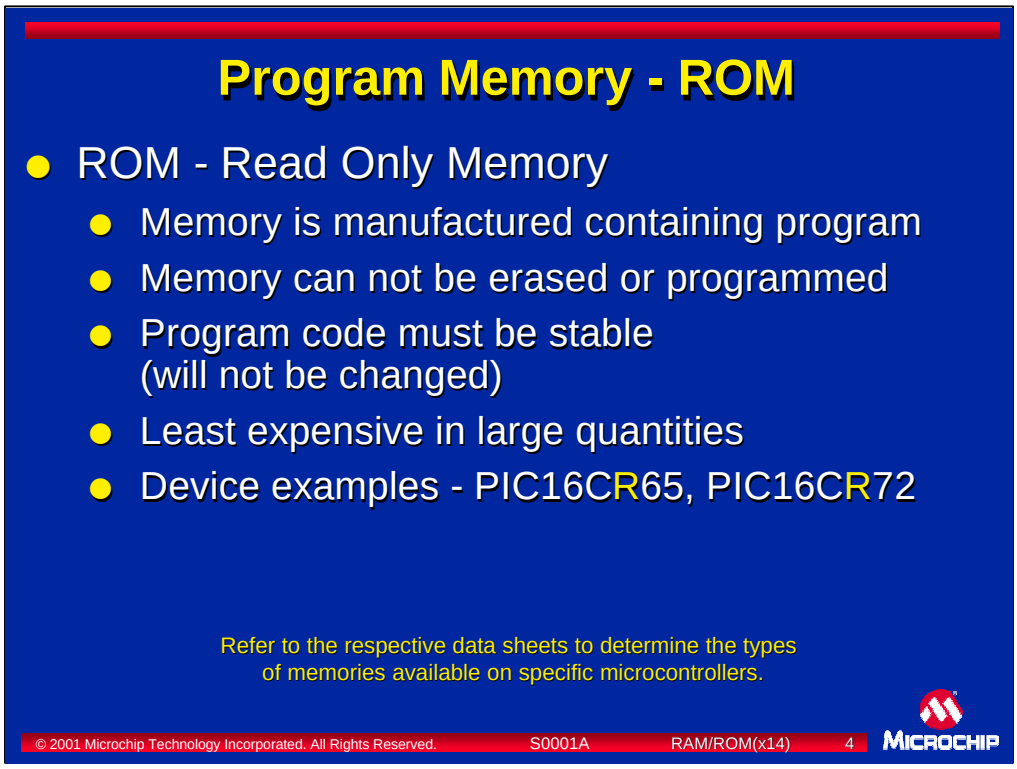


First we'll take a look at the Program Memory space. Within the program memory space there are several available memory technologies, each of which will be explained in greater detail on the following slides.

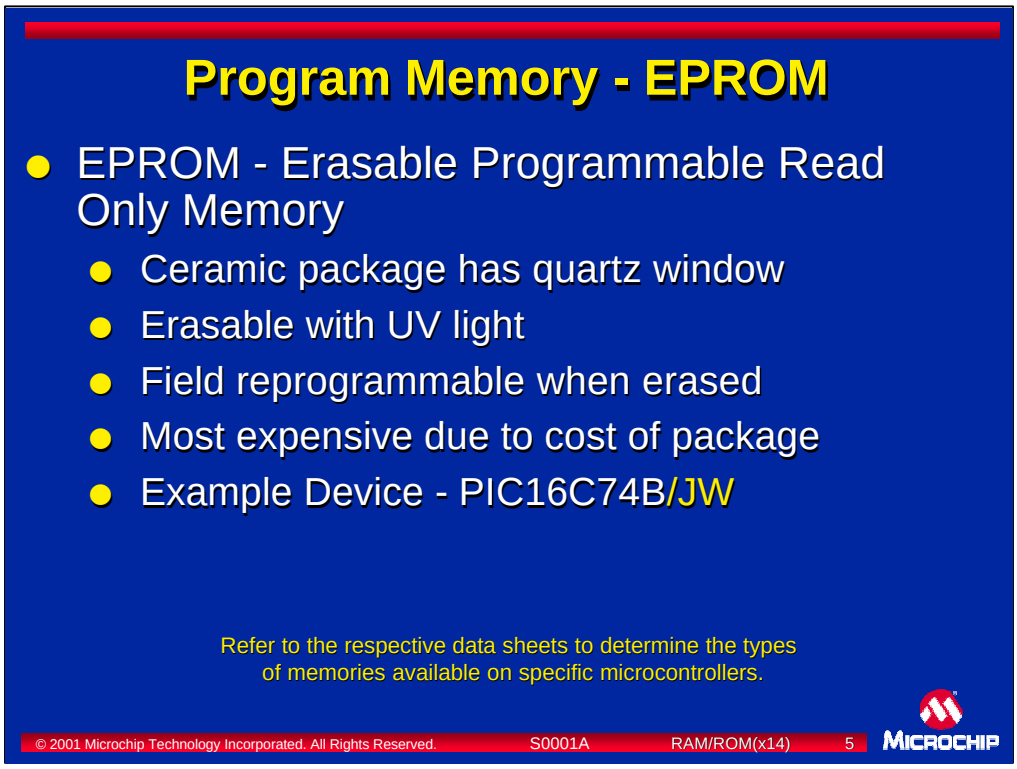
ROM memory is the least expensive program memory but is only recommended when the application code is stable and a high volume of devices are needed.

EPROM and OTP memories actually use the same die, but differ in how they are packaged. The packaging affects the cost and how a device is used. A die is the silicon "chip" inside the package that contains all the electronics.

FLASH memory devices have very fast erase/write cycles which allows for fast code development. The FLASH memory devices may also offer non-volatile data memory.

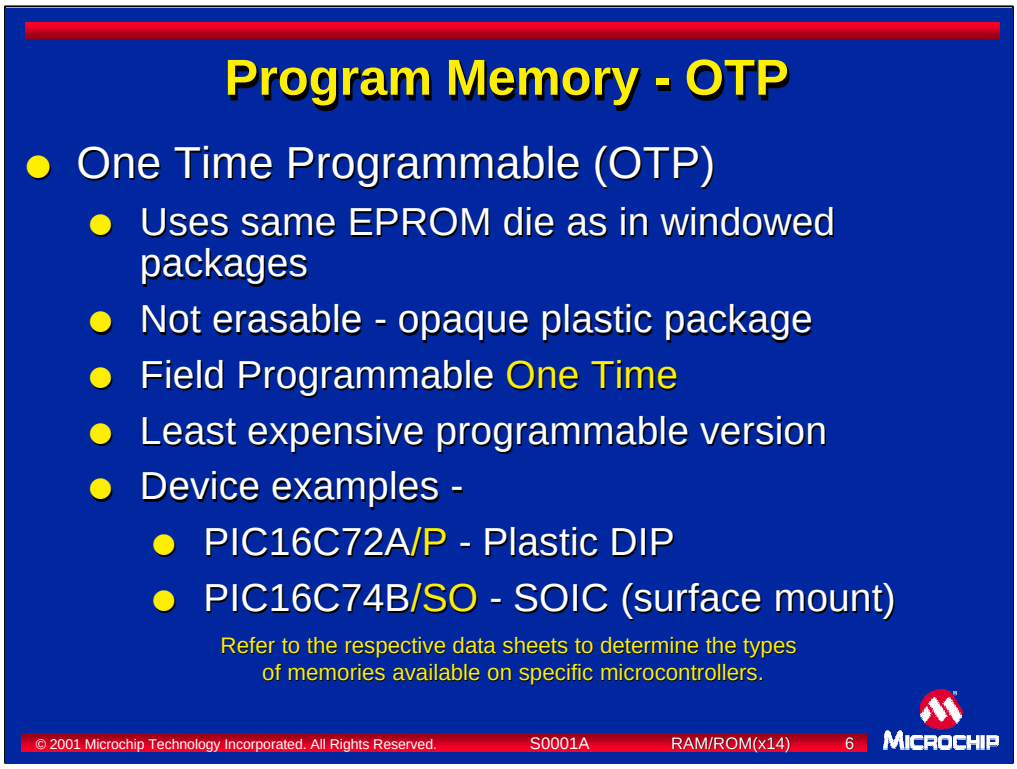


Let's take a closer look at the ROM or Read Only Memory. Microcontrollers with ROM program memory are manufactured with the desired program code already on them which cannot be changed after they have been manufactured. For this reason, microcontrollers with the ROM program memory technology are best suited in applications where the program code will not change and high volumes of the devices are required. To be cost-effective and less expensive than microcontrollers with OTP or FLASH program memory, the devices with ROM program memory must be ordered in large quantities. Devices such as the PIC16CR65 and PIC16CR72 have ROM program memory and are denoted with an "**R**" in the part number.

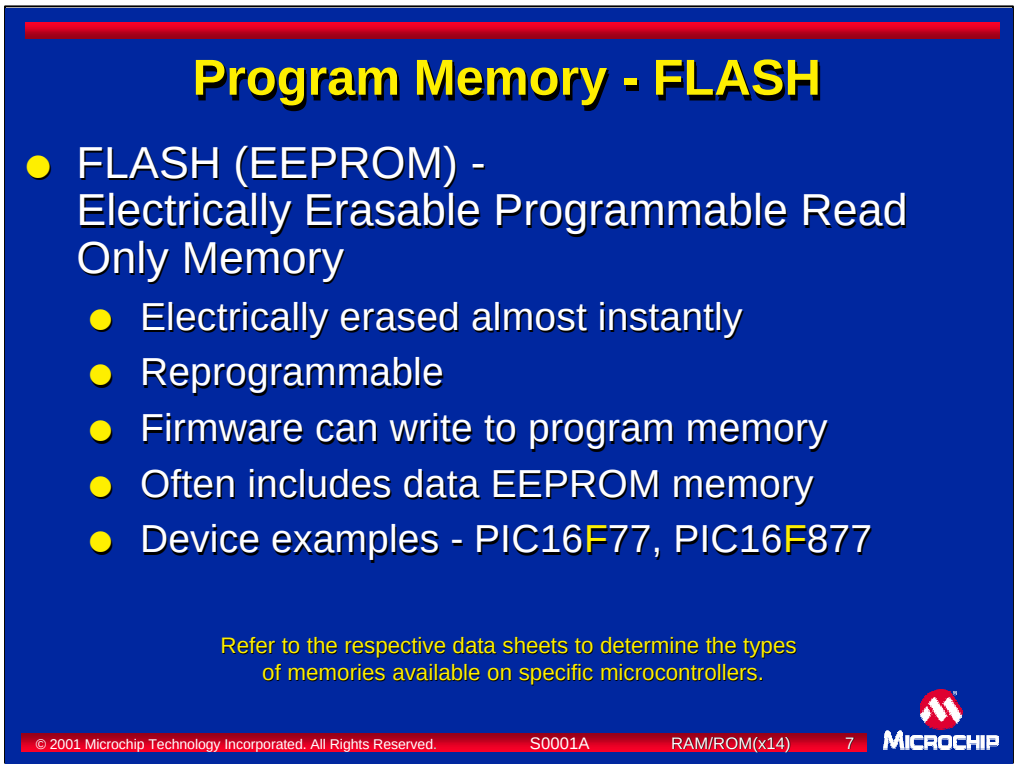


The second type of program memory is actually used in 2 different package types. This memory type is the Erasable Programmable Read Only Memory or EPROM.

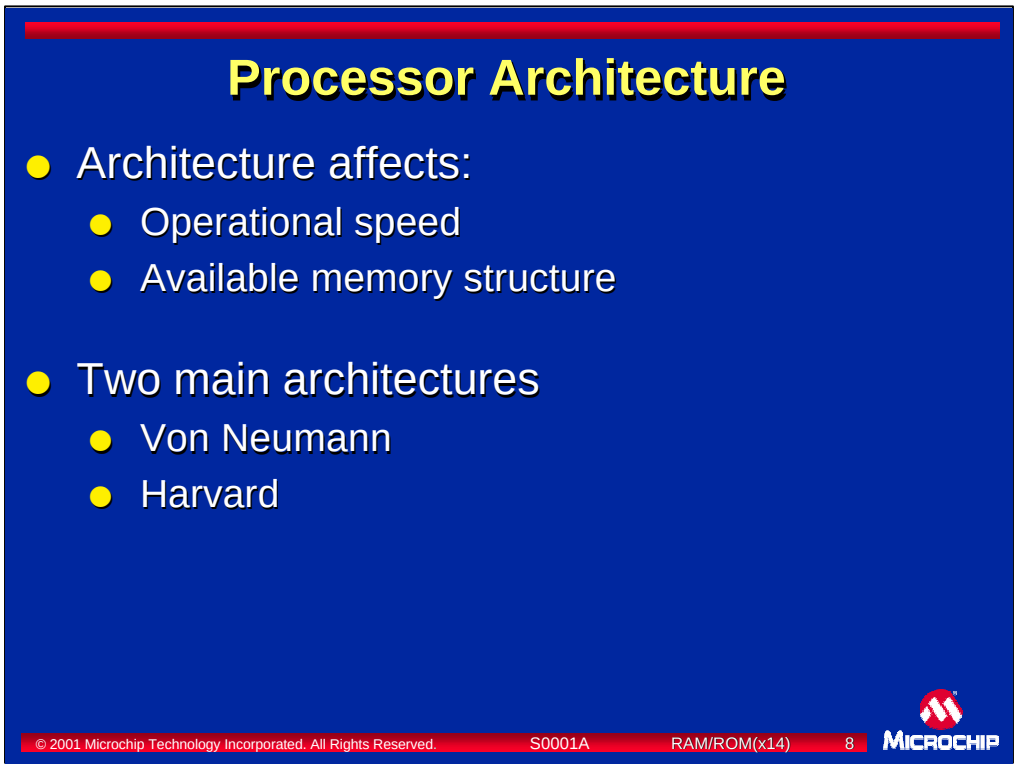
When an EPROM die is mounted in a ceramic package with a quartz window, the microcontroller can be erased using an ultraviolet eraser and reprogrammed many times. Erase times depend on the light intensity, light wavelength, the age (operating time) of the light source, and the device being erased. Typical erase times range between 5 and 30 minutes. EPROM is the most expensive version of program memory due the the high cost of the windowed ceramic package. Devices such as the PIC16C74B/JW are of the EPROM type and are available in a windowed package. The "/**JW**" suffix denotes the windowed package.



The One-Time-Programmable (OTP) microcontrollers actually use the same die as the windowed-package EPROM devices. It is the packaging that makes them unique. Since the OTP microcontrollers are in an opaque plastic package, they cannot be erased using UV light. OTP devices are shipped to the customer“blank” from the factory, and can then be programmed only once. This is why they became know as as “One Time Programmable” or OTP devices. This is the lowest cost programmable version of a device. OTP dvices such as the PIC16C72A/P (plastic DIP) and PIC16C74B/SO (SOIC surface mount) are of the EPROM type and are denoted by a suffix other than “/JW” such as “/P”, “/PQ”, “/SP” and others. Check the datasheet to see which packages a particular device is offered in.

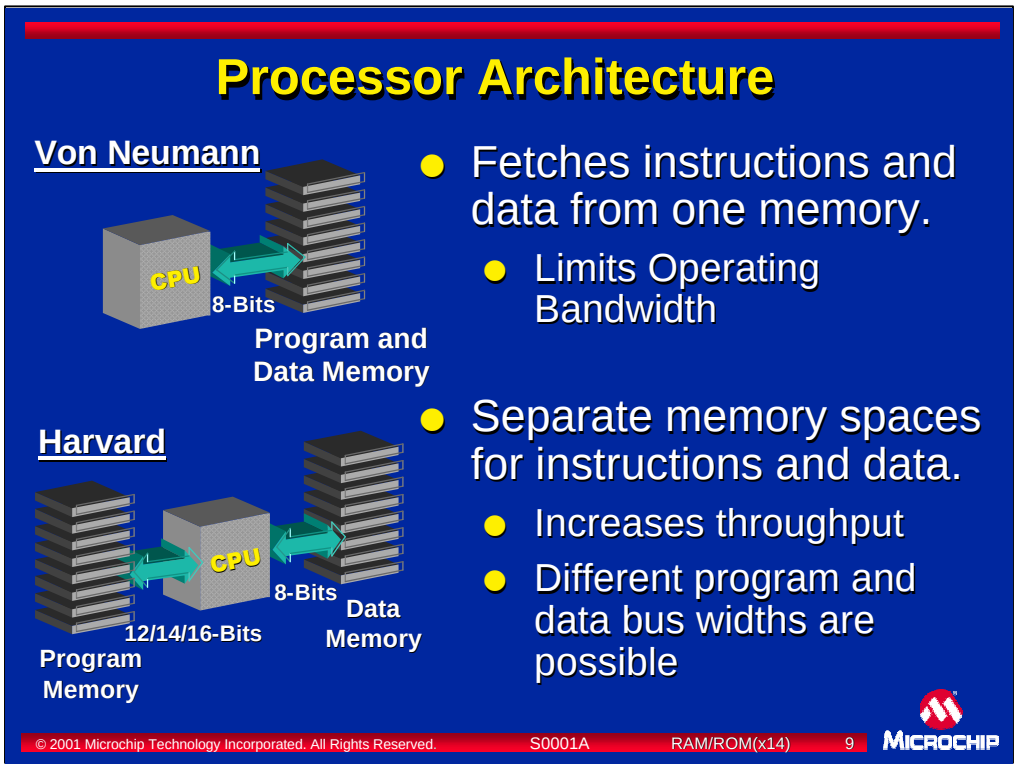


The final program memory technology we are looking at is FLASH. FLASH memory provides the ultimate flexibility because it can be electrically erased by a programmer in just a few seconds and reprogrammed. UV erasure is not required, and is not possible. Once erased in a programmer, FLASH devices can be reprogrammed with new code. Some devices with FLASH can also self-program using a specific sequence of instructions. These devices often include a small amount of non-volatile data EEPROM memory that can be rewritten many thousands of times. Data EEPROM memory will be discussed in greater detail later in this presentation. Devices such as the PIC16F77 and PIC16F877 use FLASH program memory, and are denoted with an "**F**" in the part number.



Now that we've reviewed the various on-chip memory spaces and technologies available, it's time to take a look at the processor architecture. The architecture governs the memory sizes and structure, and ultimately, operational speed.

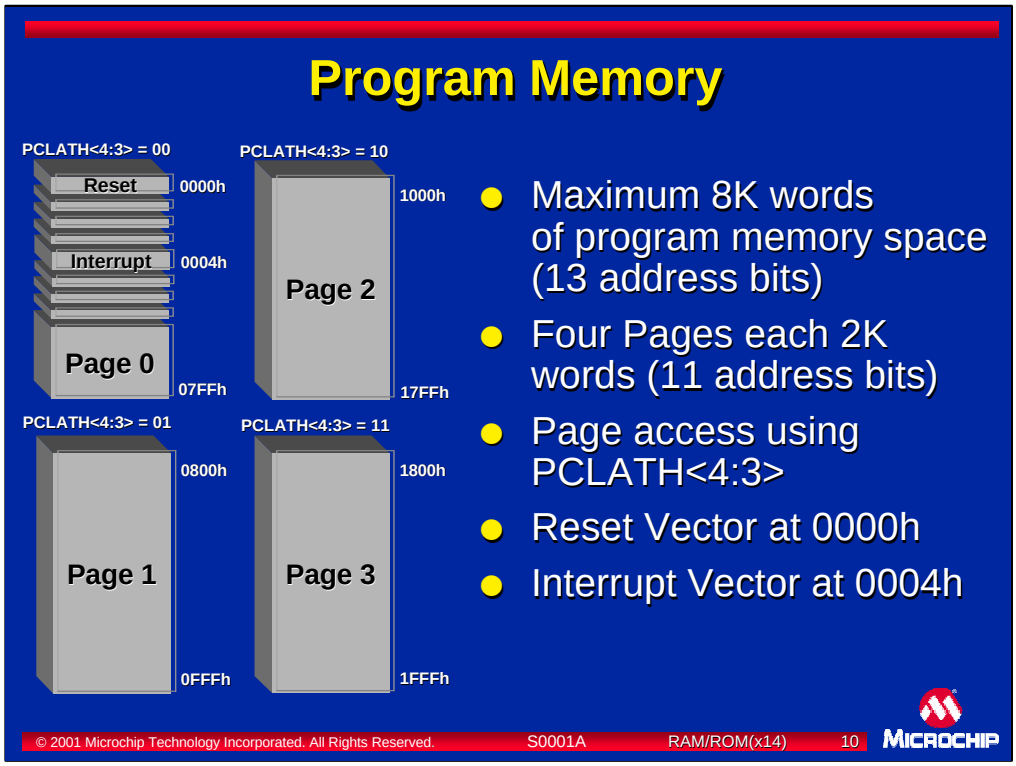
The two most common microcontroller architectures you will find are the Von Neumann architecture and the Harvard architecture.



Microcontrollers with the Von Neumann architecture have a single memory space that stores both the program instructions and the stored data. For this reason, executing instructions means that several “fetches” from the single memory space must occur. Frequently, instructions require several fetches as they cannot fit in one memory location. The first fetch retrieves the CPU instruction. Additional fetches must then retrieve data required for the program instruction. This decreases the operating bandwidth of the microcontroller because “fetching data” must wait until “fetching instructions” has completed. This is known as the Von Neumann bottleneck.

PICmicro MCUs use the Harvard architecture which has separate Program Memory and Data Memory. This allows simultaneous fetching of instructions and fetching of data in a single fetch operation resulting in increased throughput.

Another advantage of Harvard architecture is that the program and data bus widths can also be tailored to the performance requirements. While the data bus is always 8-bits wide, Microchip offers microcontrollers with program memory bus widths of 12-, 14- and 16-bits. Increasing bus widths allows greater numbers of instructions while still allowing fetching an instruction in a single fetch operation.

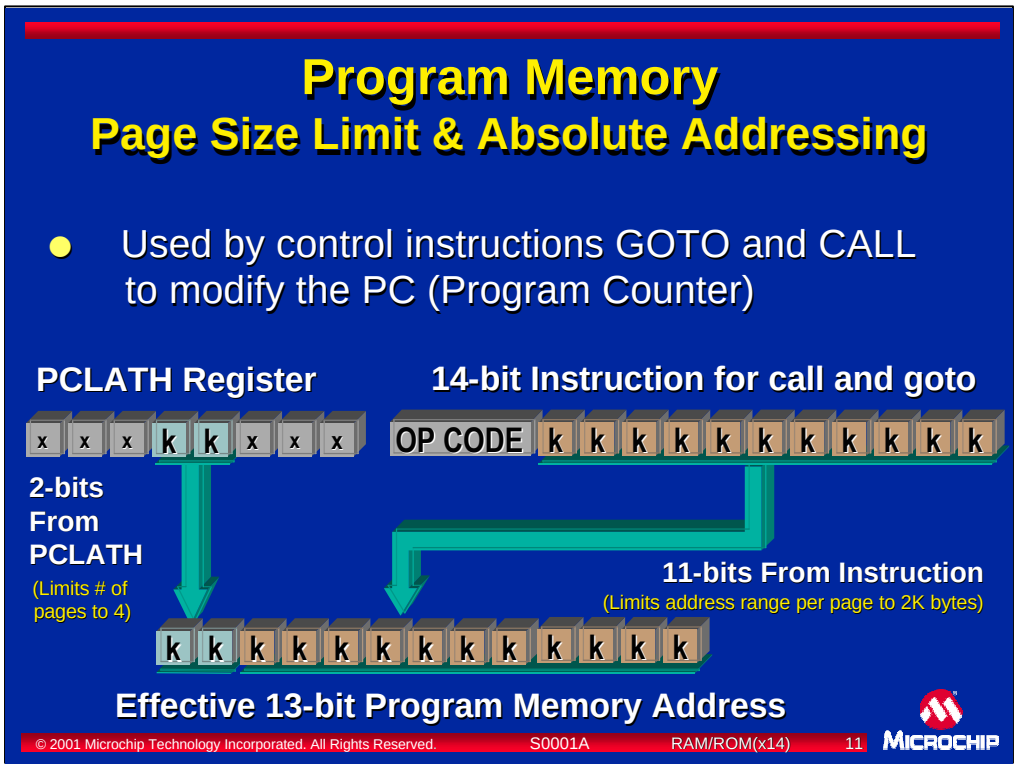


For example, the program memory for the 14-bit core microcontroller has a limit of 8K words, and each word contains a single 14-bit wide instruction. The program memory is divided into 1, 2, or 4 pages of 2K words each.

In our example, the microcontroller uses its data memory (Register file) to move among the pages. The PCLATH register is used to select which page the next execution branch will go to. When a GOTO or CALL instruction is executed, PCLATH<4:3> is used to select the page branched to. When the PCL is modified by user code, PCLATH<4:0> is used with PCL to form the full PC address of the next instruction to be executed.

As you can see on this diagram, Page 0 contains the Reset vector at location 0 (0000h). After a reset, code execution begins at the reset vector. Ordinarly, the first 2 instructions set PCLATH to the correct program page, and the third instruction is a GOTO that causes code execution to branch to another place in program memory. Otherwise, little useful code can be placed at the start of Program Memory if interrupts are used.

When an interrupt occurs during code execution, the next instruction address to be fetched (whatever it is) is saved to the stack and execution branches to the interrupt vector at location 4. Often, instructions to load PCLATH and a GOTO at this location will cause execution to branch to somewhere else in memory. Alternatively, servicing the interrupt can begin at this vector location.

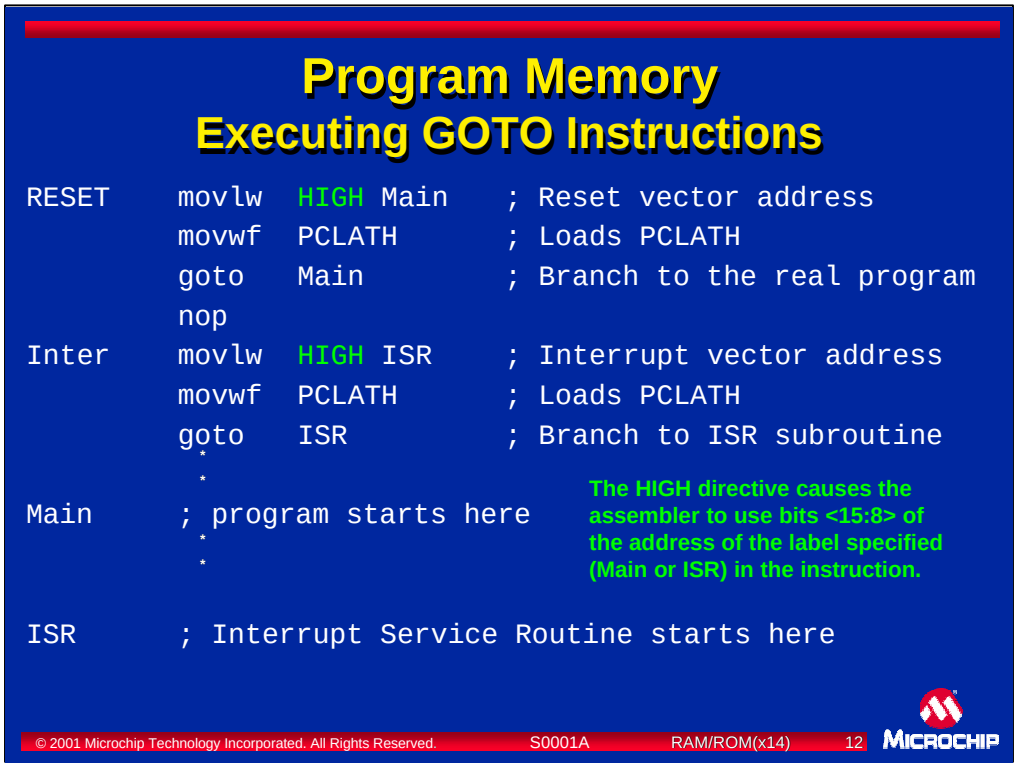


Program memory page size is dictated by the number of addressing bits encoded into a branch instruction such as the CALL or GOTO instructions.

The first 3 bits of these branch instructions indicate that this instruction will modify the program counter. In the case of the CALL instruction, they also indicate that a return address, the address of the next instruction, should be saved on the stack. The remaining 11 bits are loaded into the 11 least significant bits (LSb) of the program counter. Using 11 bits for addressing allows up to 2K of addresses. This defines the size of the program memory page.

For devices with up to 2K of program memory, the 2 Most Significant bits (MSb) of the program counter are maintained clear by keeping PCLATH clear. Devices with 4K of program memory will require keeping the 5th bit in PCLATH clear while operating the 4th bit to select one of two pages. Devices with 8K of program memory will require operating the 4th and 5th bits in PCLATH to select 1 of 4 pages. In the Mid-range PICmicro MCUs, the 3 MSbs of PCLATH are never used.

Because the entire address is explicitly defined using PCLATH and an address encoded in the instruction, we say that the we are using absolute addressing. Before a GOTO or CALL instruction is executed, the user must ensure that the PCLATH register bits 3 and 4 are pointing to the required page in program memory. If this is not done, execution will branch to the corresponding address in the currently selected page.



This is a code example for correctly executing GOTO instructions.

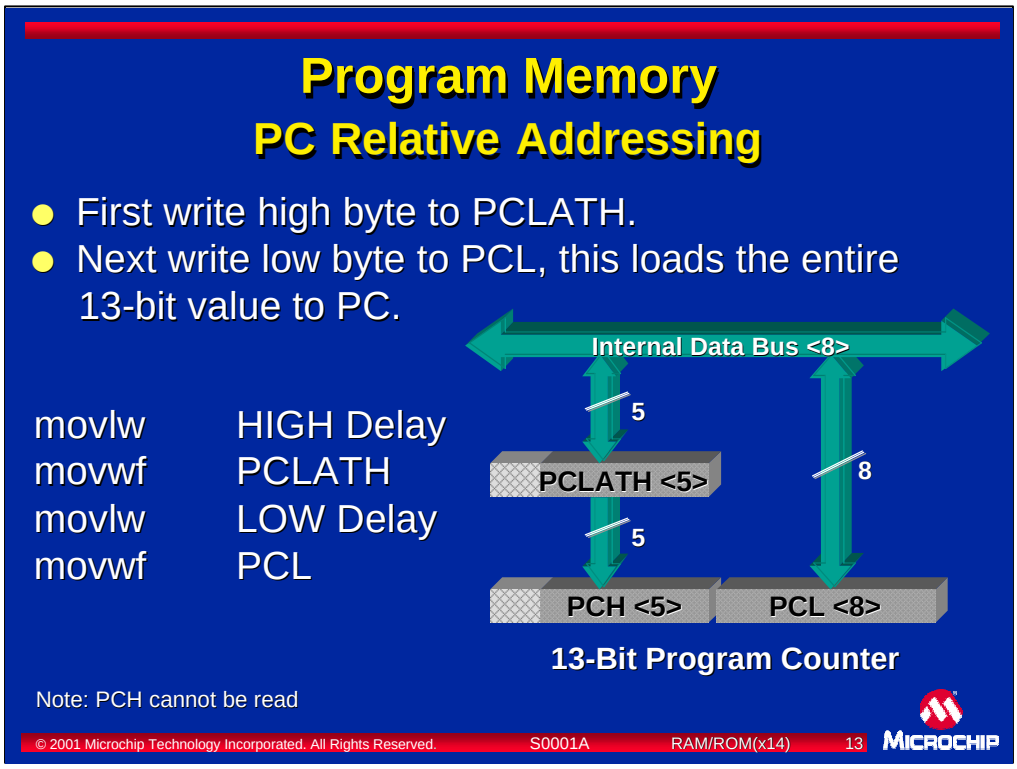
The word HIGH in the first line of code is an assembler directive that uses bits <15:8> of the addresses for Main and ISR. The instruction takes the selected 8 bits and loads them into the W register. The second instruction puts the contents of the W register in PCLATH to prepare for a branch to another page. The GOTO instruction actually causes execution to branch to the label Main. PCLATH<4:3> are used to select the required program memory page when the GOTO instruction is executed.

The NOP instruction simply occupies the only remaining program word before the interrupt vector begins, and is never executed. This location does not even have to be programmed.

The interrupt vector functions the same way as the reset vector.

The labels Main and ISR can be located anywhere in program memory.

Loading PCLATH is not always required. If all program code fits into one memory page, or the microcontroller has only one page, loading PCLATH is not required. However, program code is often spread across several pages. A GOTO or CALL can result in a branch to a page other than the one currently selected, in which case, PCLATH will need to be loaded with the required page. The absolute list file can be examined to determine which page is being jumped from, which page a label is in, and possibly the current contents of PCLATH to decide whether or not loading PCLATH is required.

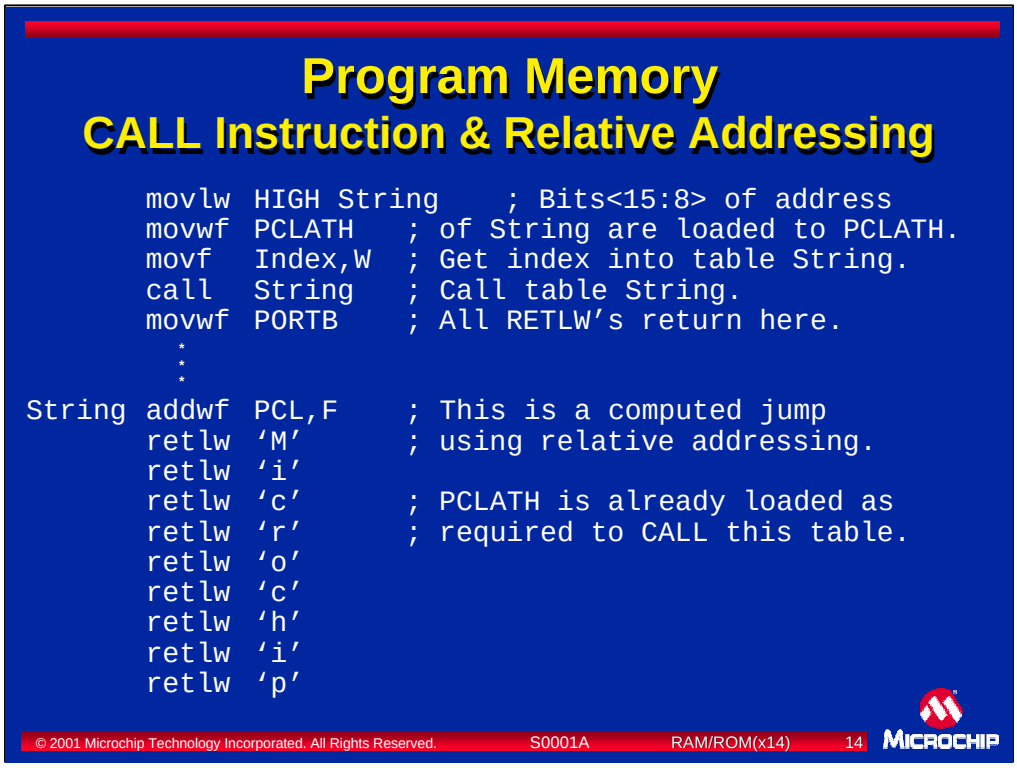


There is another way to execute a program branch when an instruction writes an 8-bit value to PCL. This can be the result of any operation where the PCL is the destination for the result. This forms the basis for what is called the “computed jump.”

The low byte of the program counter can be read and written directly using PCL. The high byte (PCH) is only writable through PCLATH. PCLATH is loaded by firmware with bits <15:8> of the address prior to writing to PCL. When PCL is modified by user code, the contents of PCLATH are also transferred into the high byte of the program counter, PCH. Only the lower 5 bits of PCLATH actually have any meaning since only 13-bits of addressing are used in mid-range devices

In this example, PCLATH is loaded with bits<15:8> of the address of Delay. The LOW directive takes bits<7:0> of the address of Delay and places them in the W register. When the W register is written to PCL, the contents of PCLATH are also written to PCH forming the complete 13-bit address to Delay. The next instruction executed is the first instruction of Delay.

PCH cannot be read, and cannot be written to directly. When PCL is written, the contents of PCLATH are written to PCH.

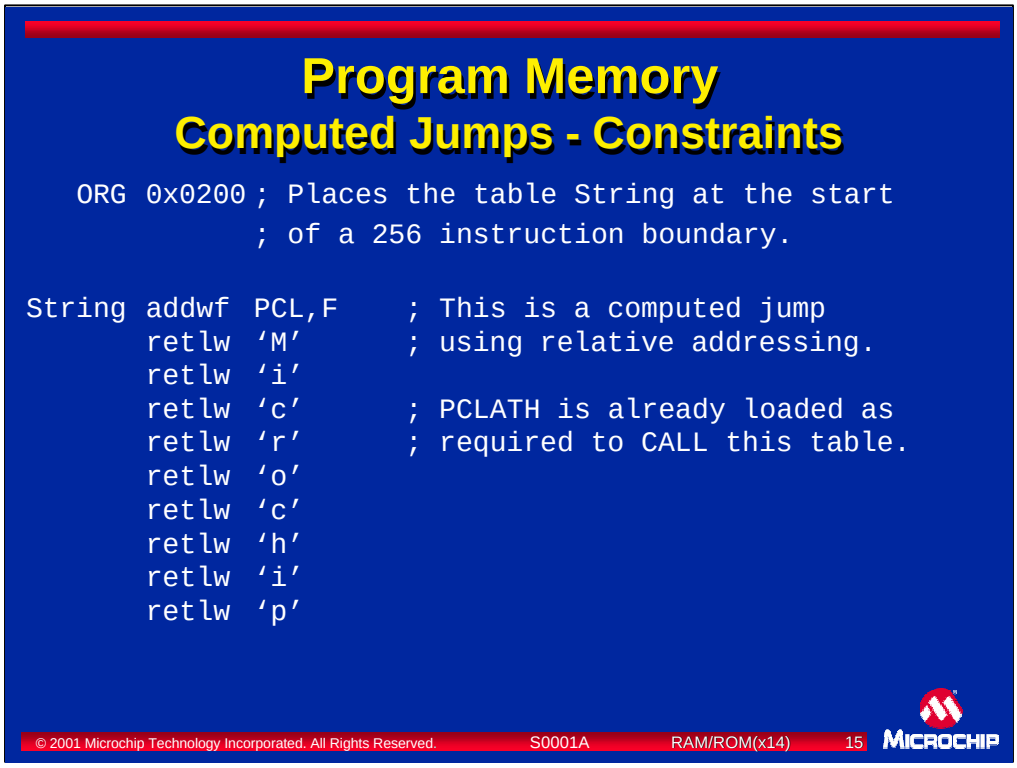


This code example shows how to correctly execute a CALL instruction, and how relative addressing works. A look-up table is implemented using a computed jump which contains the word “Microchip.” On execution of this code example, one character of the string “Microchip” will be written to PORTB. The variable Index is an offset from the start of the table, and points to one of the characters in the table.

The HIGH directive tells the assembler to use bits<15:8> of the address to String for the MOVLW instruction. The next instruction loads the contents of the W register into PCLATH. Only bits 3 & 4 of PCLATH are used when a CALL is executed. The offset into the table String is loaded into the W register and a call to String is made.

The first line of String adds the value in PCL to W. The result is saved back to PCL as specified by the “,F” at the end of the instruction. When the ADDWF instruction is executed, the program counter is already pointing to the next line of code, in this case, RETLW “M”.

Assuming that Index was set to 3, when the ADDWF instruction completes, a NOP is executed in place of the RETLW ‘M’ instruction already fetched. The next instruction fetched is the the RETLW ‘r’ instruction pointed to by the new PC. This takes two cycles, one to execute a NOP in place of the previously fetched instruction and one to fetch the new instruction.



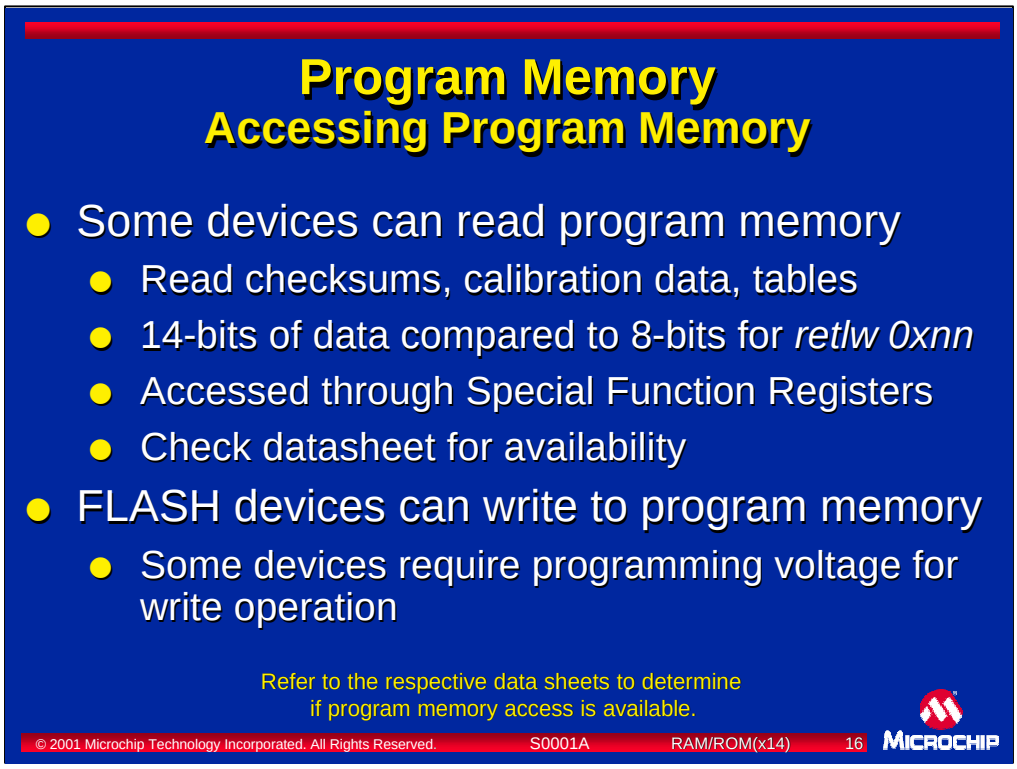
There are a few constraints on the size and location of computed GOTO tables.

When the index is added to the PCL, and an overflow occurs from the addition, it will not cause an increment of PCH or PCLATH (the overflow is lost). This means that such tables should be completely contained between 256 instruction boundaries, as an attempt to jump to an address outside the current 256 instruction boundaries will fail. The address jumped to will be formed using the current PCLATH and resulting PCL.

The ORG directive causes the assembler to locate the next instruction at the address specified. In this case, the ADDWF instruction is placed at 0x0200 which is a 256 instruction boundary.

Because the ADDWF PCL,F instruction must also be included within the boundaries, the number of entries is limited to 255.

Several smaller tables can be implemented between the same pair of adjacent boundaries if they are all completely contained by those boundaries.



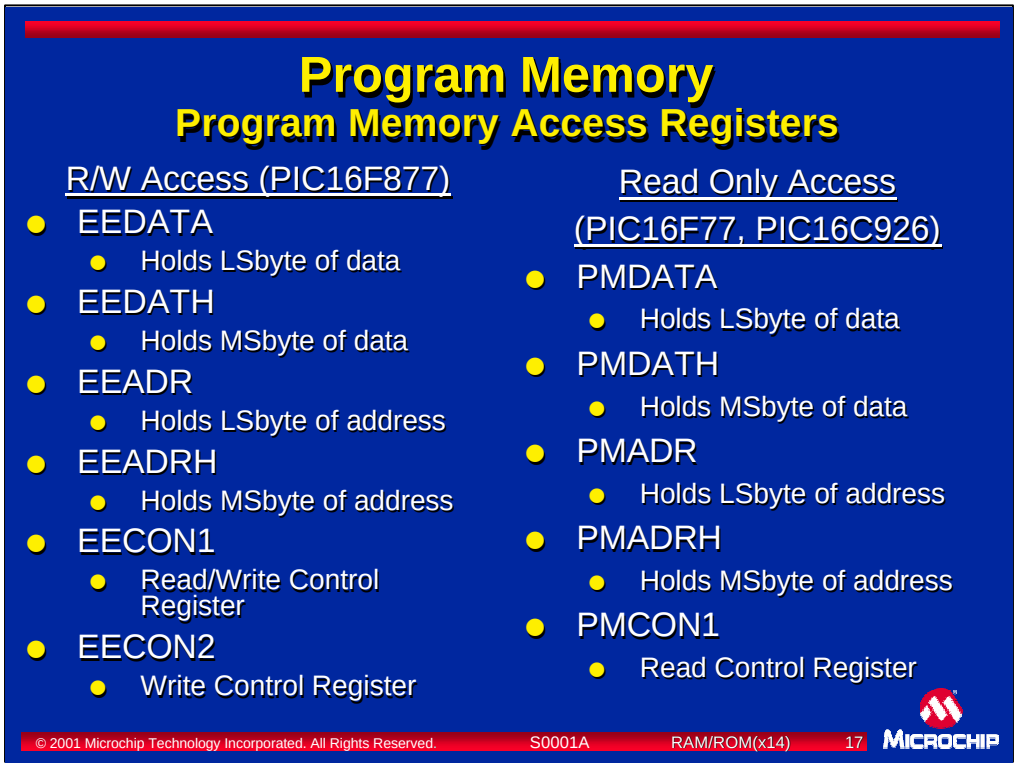
Some devices can read directly from program memory. This capability makes possible the reading of program memory to calculate checksums, retrieving calibration data or using large look-up tables.

The program memory access function allows 14-bit data to be stored directly in memory (such as calibration data.) This data can be any value and does not have to be a valid instruction. If the data in a program memory location does not form a valid instruction and is executed, the result will be a NOP instruction which does not alter the state of the microcontroller. This function allows more optimized storage of information when compared to retlw instructions which allow only 8-bits per word of program memory.

A set of six special function registers control what memory address is accessed. Two registers are used to select the address, two are used to present the program memory contents for use as data, and two are used to control memory accesses.

Check the datasheet for your microcontroller to see if this function is offered.

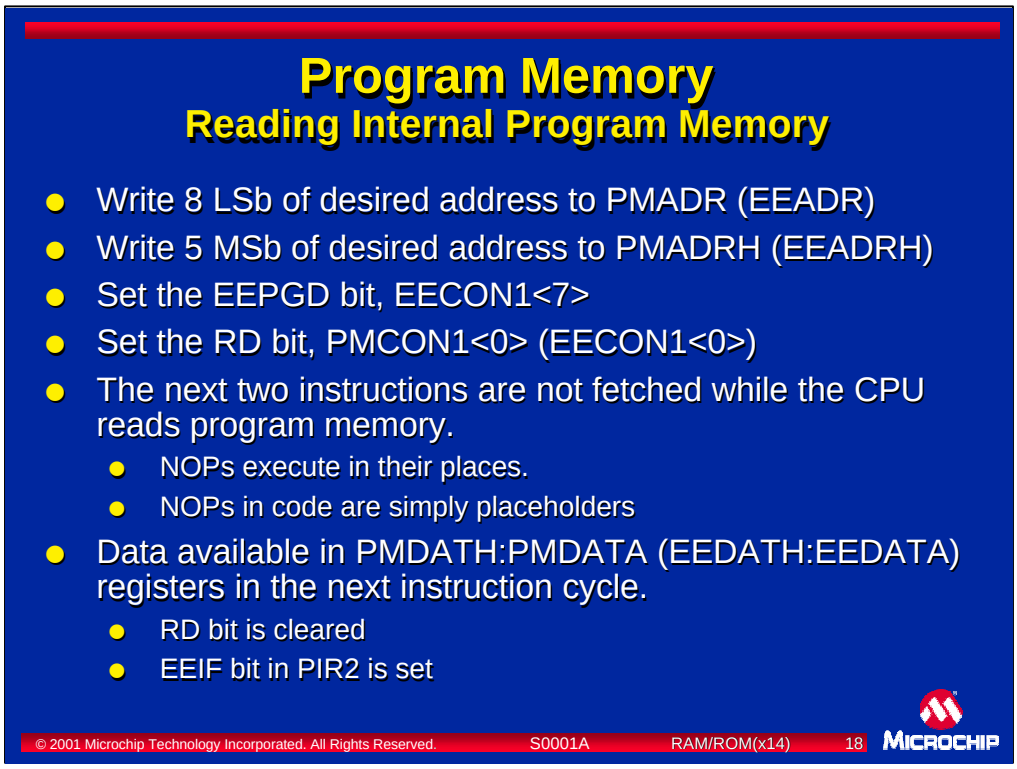
FLASH devices can also write to program memory. Many FLASH microcontrollers can perform writes using only V_{DD} as the supply voltage. Others require programming voltage for writes.



Members of the PIC16F87X family have both program memory read and write access. The special function register set used to access program memory is the EEDATA and EEDATH register for holding 14-bit data values, EEADR and EEADRH for holding the 13-bit program memory address, and EECON1 and EECON2 for controlling the read or write operation.

Some devices with read only access include the PIC16F7X family and the PIC16C925 and PIC16C926 microcontrollers. These devices have the following special function registers: PMDATA and PMDATH for holding 14-bit data, PMADR and PMADRH for holding the 13-bit program memory address, and PMCON1 for controlling the read.

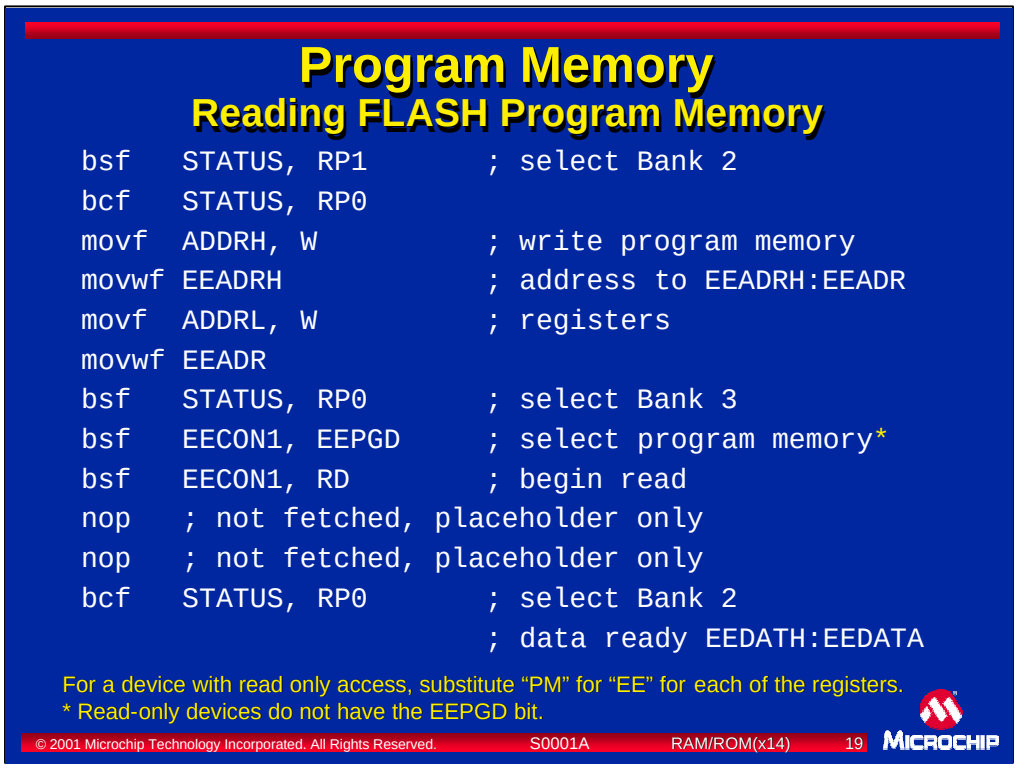
Please refer to the specific microcontroller data sheet for details.



The process of reading internal program memory is straight forward. The address of the desired program memory location is loaded into the PMADRH:PMADR registers (or EEADRH:EEADR registers for read/write devices). If using a device with read/write functions, the EEPGD bit in the EECON1 register needs to be set to indicate the access must take place in program memory.

To initiate the read operation, the RD bit in the PMCON1 (or EECON1) register is set. The value of the desired program memory location is automatically read during the next two instruction cycles and placed in the data registers. The two instructions that follow setting the RD bit are not fetched and NOPs are executed in their places. The RD bit is automatically cleared after the operation completes, and the EEIF flag in PIR2 is set. The data registers contain the contents of the desired program memory address.

The contents of the address registers are not modified by the read operation. The data registers continue to hold their values until firmware modifies them.



This is a source code example for reading the program memory on a microcontroller with read and write access. In a device with read only access, the names of the SFR registers are changed to their respective PM register names, and the `BCF EECON1,EEPGD` instruction is removed.

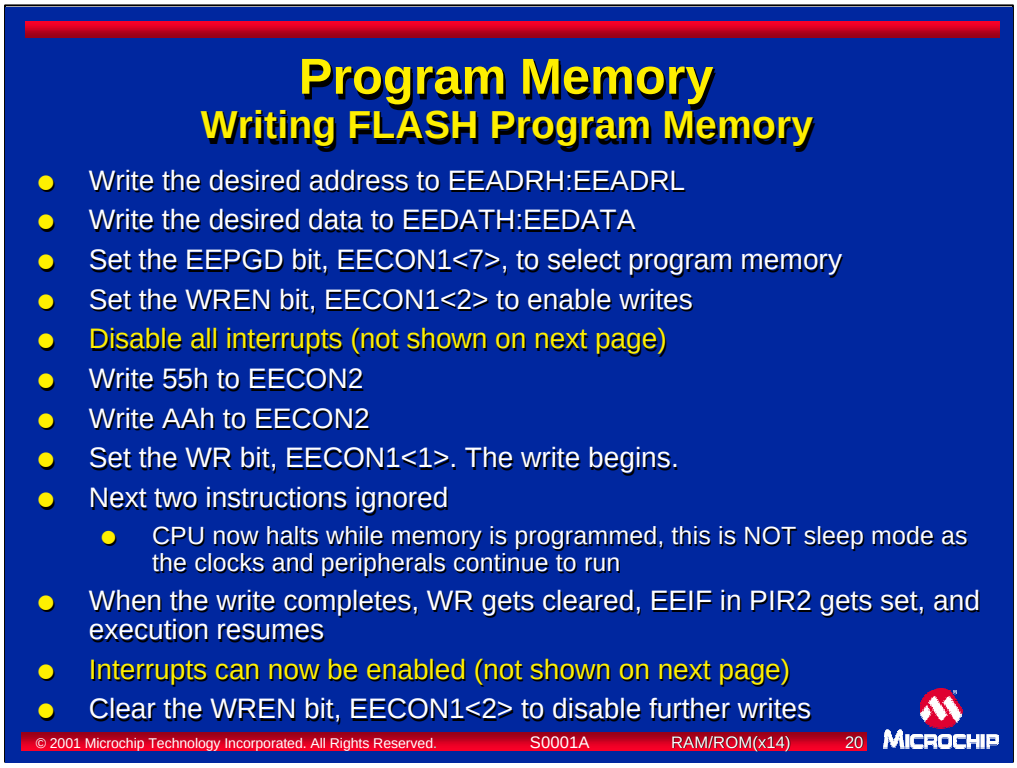
You will notice that two NOP instructions follow the setting of the RD bit. These two instructions are not fetched, but are present as placeholders. NOPs are executed in their place.

RP0 and RP1 are operated to select Bank 2 of data memory. The address to be read is contained in ADDRH:ADDRL. The high byte of the program memory address to be read is loaded into the W register. The W register is then loaded into EEARDH. This is repeated for the low byte of the address.

RP0 is set to select Bank 3 where EECON1 can be accessed. The EEPGD bit in EECON1 is set to point to program memory. The RD bit in the same register is then set. During the next 2 instruction cycles, no program instructions are fetched and NOPs are executed in their places.

When instruction fetching and execution resume, the RD bit is cleared, the EEIF bit in PIR2 is set, and the EEDATH:EEDATA registers in Bank 2 contain the program memory word just read from the selected address. Bank 2 is selected by clearing the RP0 bit.

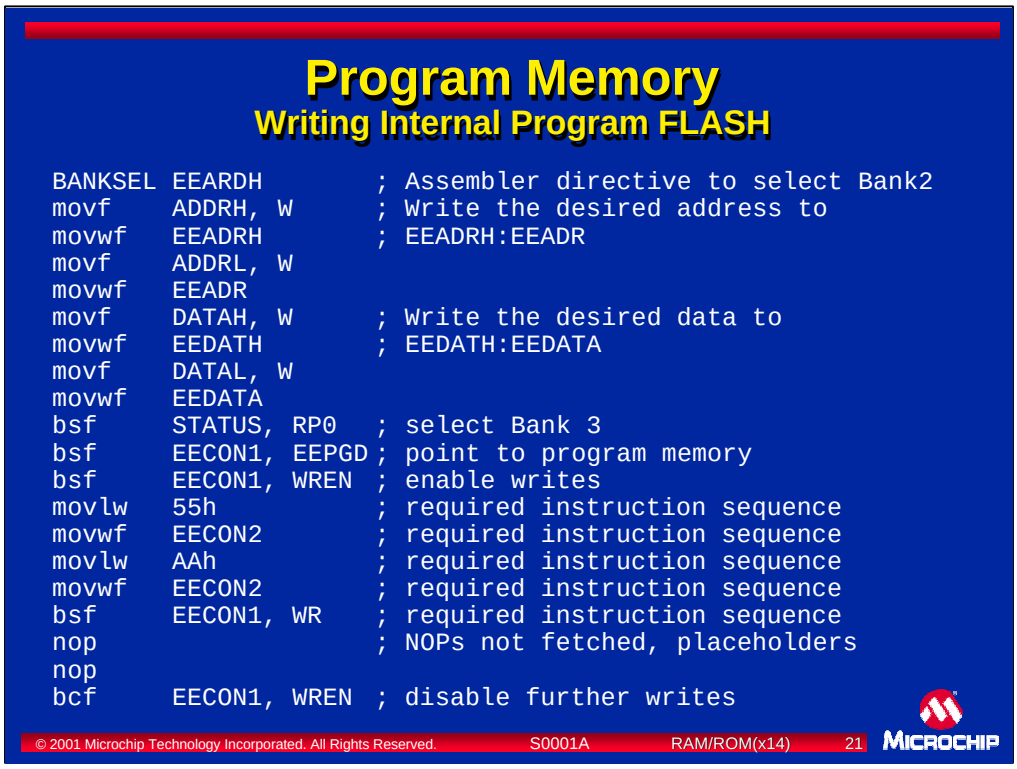
Not shown is actually reading the EEDATH:EEDATA registers.



There are more steps required to write to program memory. While both read and write operations require that the address is loaded into the address registers, the write function also includes loading the desired data to EEDATH:EEDATA, and executing a very specific sequence of five instructions to perform the write.

Interrupts should be disabled before the five instruction sequence. If an interrupt should occur during this sequence, the write will not complete. Interrupts can be re-enabled after the write operation completes.

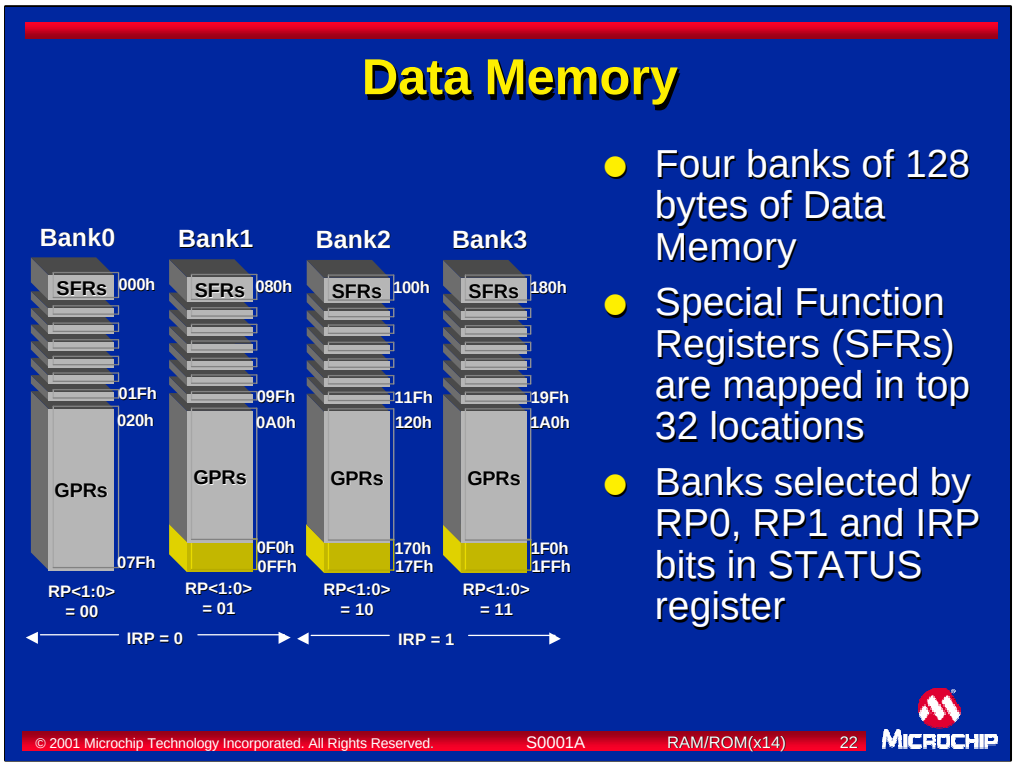
There are two safety mechanisms that prevent inadvertent writes to program memory. The first is a write enable bit called WREN. It must be set before write operations will work. The second mechanism is a special sequence of 5 instructions which must take place consecutively without any interruptions. At the conclusion of these instructions, the write will begin. When writing to program memory, the execution of instructions will stop while the data is being written. The oscillator will continue to run and all peripherals will continue to operate as configured. Interrupts will be disabled, but the interrupt flags can still be set. When the write has completed, the WR bit is cleared, the EEIF bit in PIR2 is set, and execution of code resumes. The write enable bit WREN should then be cleared to prevent unintended write operations. Interrupts can be re-enabled. When interrupts are re-enabled, the program will branch to the interrupt vector for the interrupt service routine to handle any pending interrupts.



This code example demonstrates how to write data to program memory.

The differences between reading and writing program FLASH memory are:

1) Reads retrieve data from the EEDATH:EEDATA register after the operation. Writes place data to write in these registers before the write.
2) Writes must be enabled by setting the WREN bit. There is no corresponding requirement for reads.
3) Writes require a specific 5 instruction sequence to perform the write. There are no corresponding requirements for reads.
4) Writes are started by setting the WR bit. Reads are started by setting the RD bit.
5) Execution of instructions during reads does not occur for only 2 instruction cycles. Instruction execution for writes is halted for as long as several ms. (See programming specification.)
6) When execution resumes, the WR (writes) or RD (reads) bit is cleared automatically.
6) After a write, the WREN bit should be cleared to prevent another write. There is no corresponding requirement for reads.



Now that we've reviewed the program memory functions, let's take a closer look at data memory - location of the variable data values (SFRs, GPRs). The data memory on the PICmicro MCU may have up to four banks with 128 bytes each. Please refer to the specific microcontroller data sheet for the composition of data memory banks.

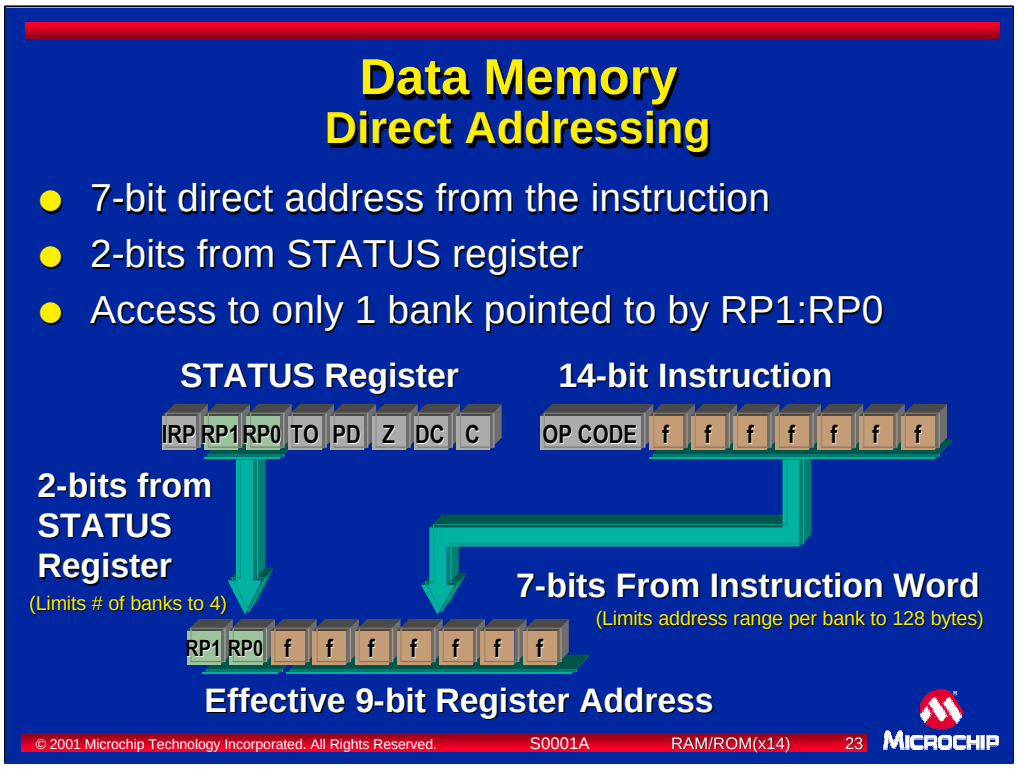
The first 32 bytes in each bank are reserved for Special Function Registers. Some of these Special Function Registers (SFR) appear in all banks such as the STATUS and File Select Register (FSR) . Some addresses in these sections have no SFR and are not implemented.

Some PICmicro devices have a shared data memory region. This memory is shared across all banks. In other words, the same memory location within a bank can be accessed in all banks without having to select a different bank. If it is present, this shared memory is usually the last 16 bytes of each bank.

Labels for variables in the shared memory region should be declared only once, in any bank, but will be available in all banks.

When direct addressing is used, the RP1 and RP0 bits in the STATUS register select the desired bank for direct memory access. Within each bank, 7 bits of addressing select 1 of 128 addresses. Therefore, in a microcontroller with 4 banks of data memory, 9 bits are required to uniquely address any data memory location.

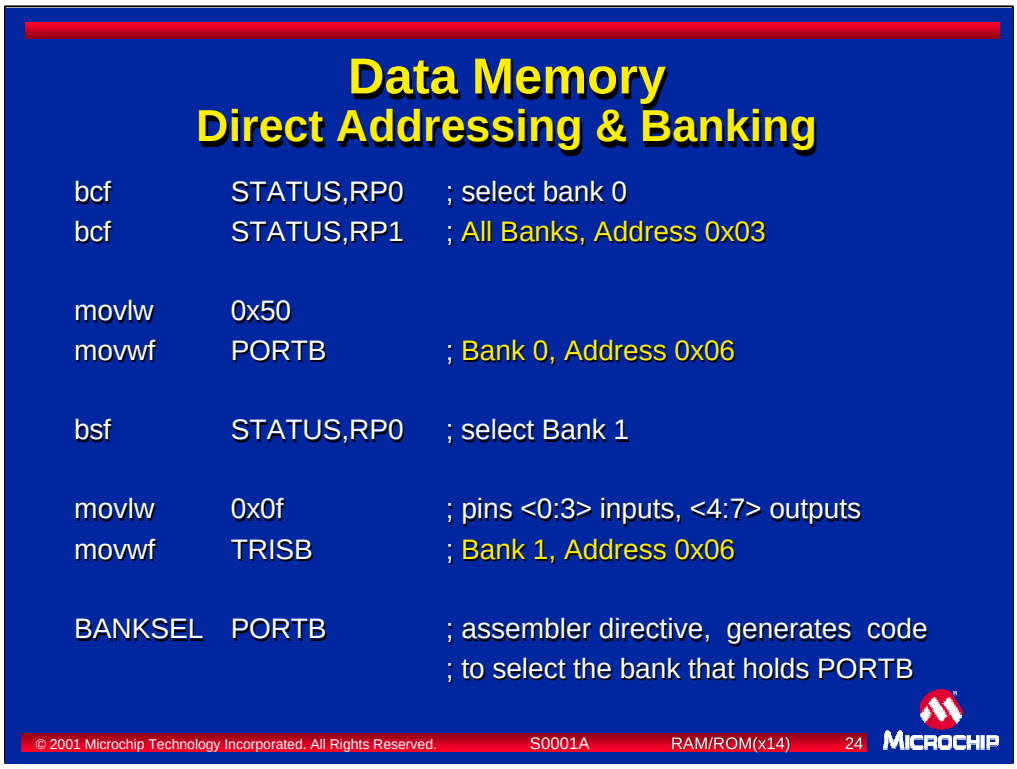
When indirect addressing is used, the IRP bit is used to select either Banks 0 and 1, or Banks 2 and 3 for indirect addressing. The FSR register provides the remaining 8 bits of address data.



To access all available data memory, 9 bits of address are required. The 7 Least Significant bits (LSb) of the address are encoded in the instruction. This allows an instruction to access one byte in the selected bank.

The upper two bits of the address select the desired bank. These bits are provided by the RP0 and RP1 bits in the STATUS register. All PICmicro MCUs have at least 2 banks, so the RP0 bit must be set correctly for the desired memory bank. Some PICmicro MCUs have 4 banks and therefore both the RP0 and RP1 bits must be set to select the desired bank.

The operation that sets the RP0 and RP1 bits is called banking, and must be performed in user code. An alternative to setting the individual bits in the STATUS register is to use the BANKSEL directive. The BANKSEL directive will generate code to set the RP1 and RP0 bits to select the desired bank.



This example shows how banking works, that some registers are in all banks, and other registers occupy the same address but are in different banks.

We can always access the STATUS register from any bank. Using it, we can select the banks that other registers are located in.

In the first 2 lines, the RP0 and RP1 bits of the STATUS register are both cleared to select Bank 0.

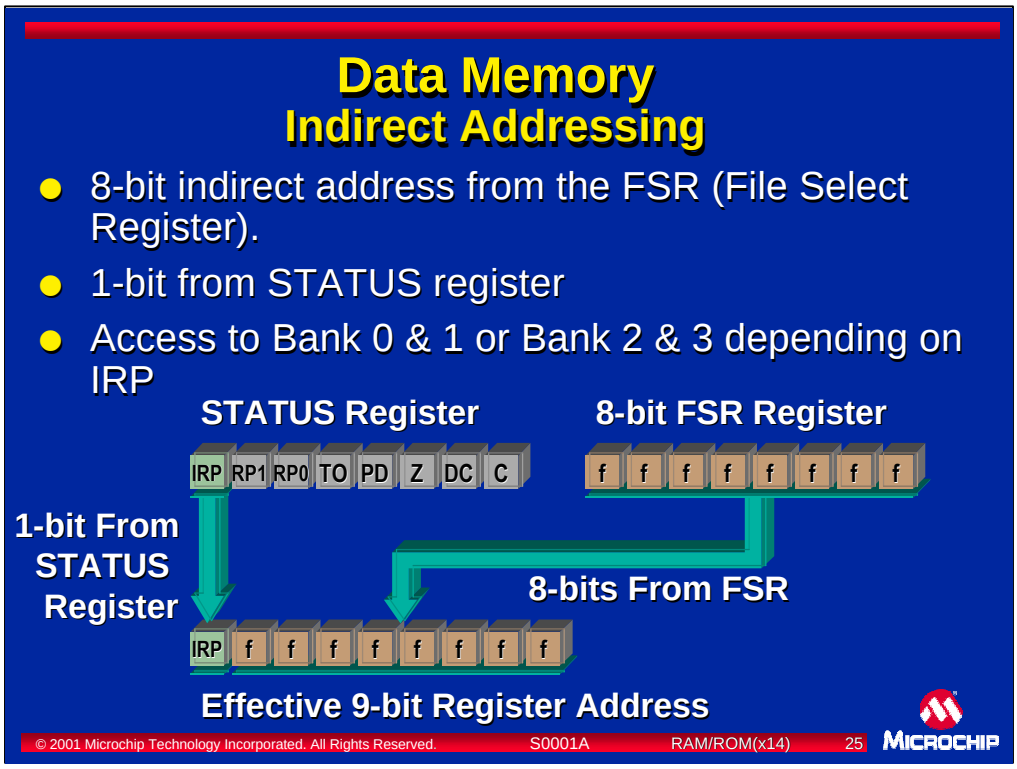
The next instruction is a literal instruction that loads some data into the W register. The contents of the W register are then moved to the PORTB register.

The next 2 lines select Bank 1, which is where the TRISB register is accessed.

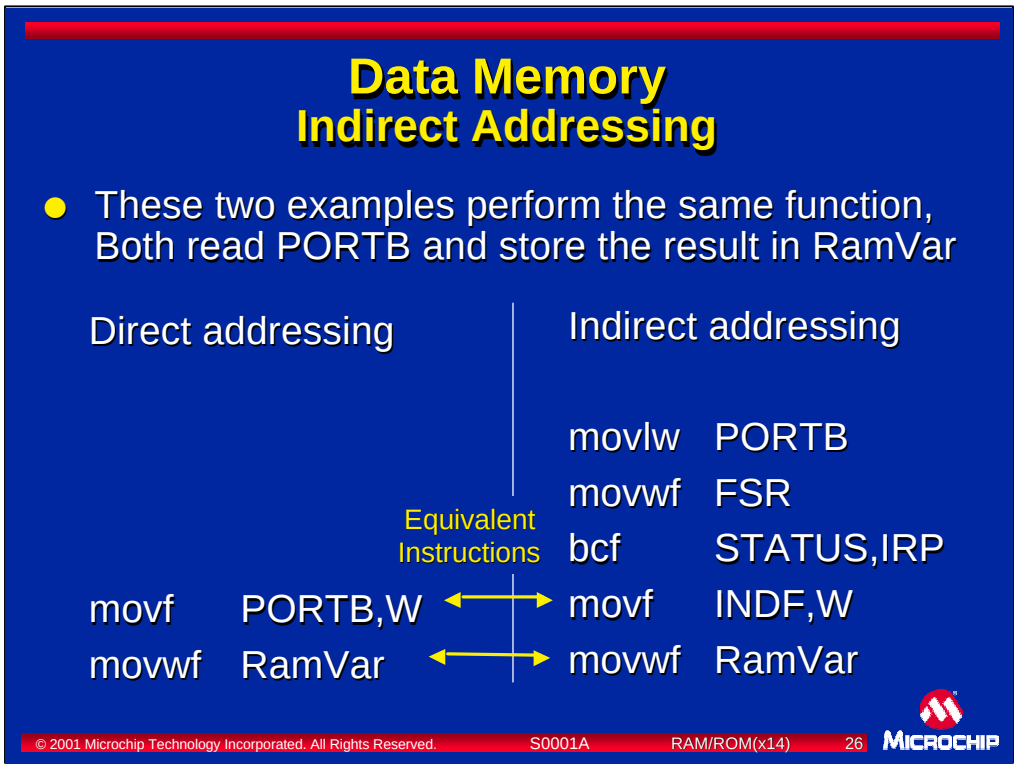
Another literal instruction loads the W register with configuration data for the PORTB pins. This data configures the PORTB pins as inputs or outputs. The contents of the W register are moved to the TRISB register.

Pins <7:4> are configured as outputs, pins <3:0> as inputs. Pins RB0 and RB2 are output high, RB1 and RB3 are output low, while RB4:RB7 remain inputs. Before the TRISB register was loaded, all PORTB pins were configured as inputs.

The last line is an assembler directive to generate code to select the bank that PORTB can be accessed in. The assembler will generate the same code as in the first 2 lines of this example.



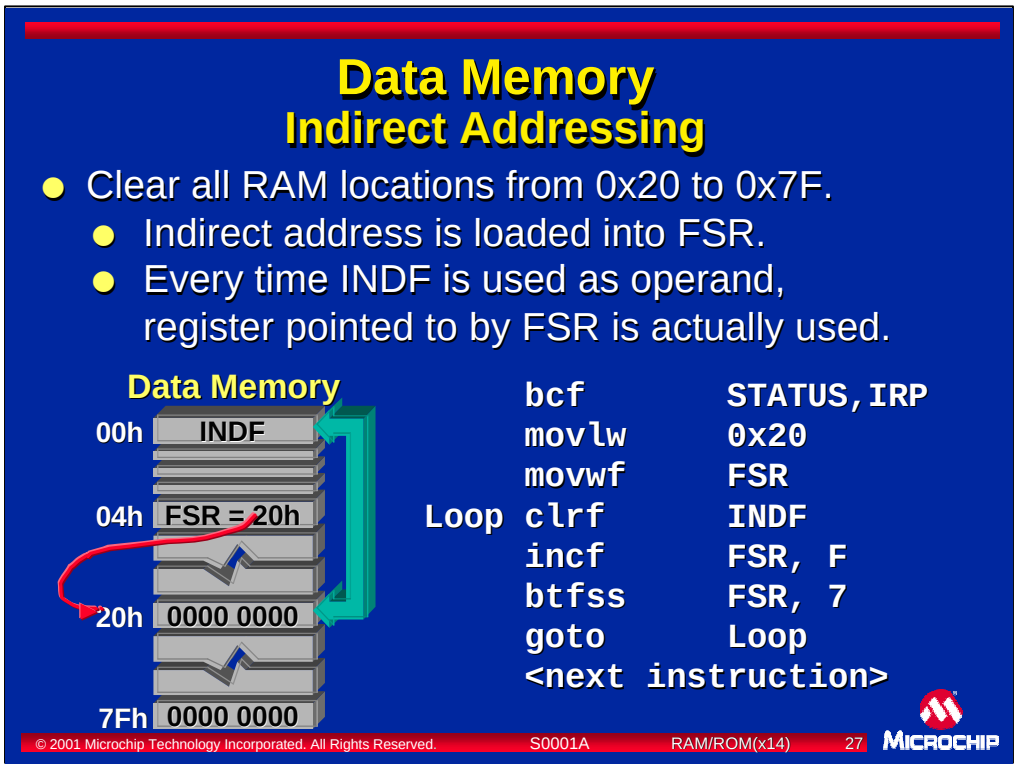
Indirect addressing uses two registers to read or write data memory locations. The File Select Register, or FSR, is used to hold the address of the desired data memory location. The IRP bit in the STATUS register selects pairs of banks. The IRP bit and the 8 bits of the FSR register are used to form the 9-bit address. In devices with 2 banks, the IRP bit should remain clear. In devices with 4 banks, the IRP bit controls whether Banks 0 and 1 are accessed, or Banks 2 and 3 are accessed.



To select a register using indirect addressing, the FSR register and IRP bit are set to the desired data memory address. The selected address is accessed by using the INDF register as the operand in any instruction. INDF is not a physically implemented register. Any access to INDF exactly mimics an access to the data memory location with the same address as specified by the IRP bit and FSR register.

For instance, if the program directly reads PORTB, the movf PORTB,W instruction is used to read the value of PORTB into the W register.

The indirect method would be to load FSR with the address of PORTB and clear the IRP bit. Then the instruction movf INDF,W would indirectly read the value of PORTB and put it into the W register.



Here is a short example of indirect addressing which clears all data memory locations in Bank 0 from address 32 to 127.

First the IRP bit is cleared to point to Banks 0 and 1. Then the file select register is loaded with hex 20 to point to the starting address.

A loop is then executed:

The contents of the register pointed to by FSR is cleared using a clrf INDF instruction.

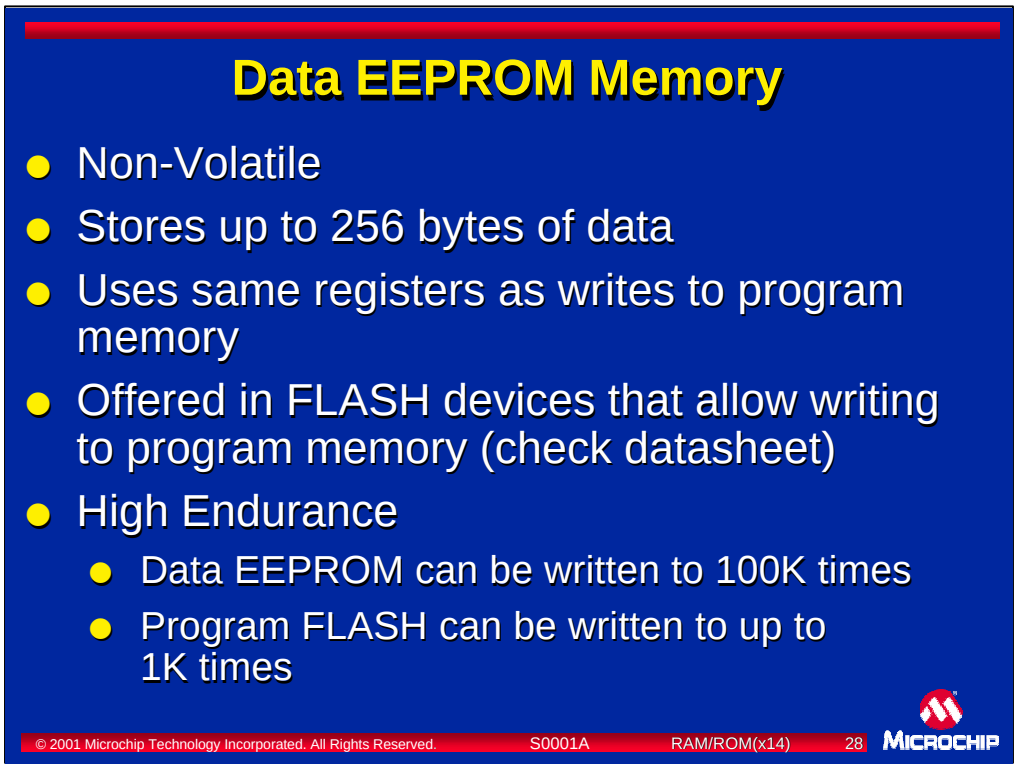
The FSR register is then incremented to the next location.

The btfss FSR,7 instruction tests bit 7 in the FSR register to determine if it is set. Bit 7 will be set only if the number is 128 or greater.

If bit 7 is clear, the program has not yet cleared all registers up to 127. The btfss FSR,7 instruction will not perform the skip and the next instruction executed will branch back to the top of the loop.

If bit 7 is set, indicating a value of 128, then the program has cleared all data memory locations from 32 to 127 and can continue. The btfss FSR,7 instruction will skip over the goto Loop instruction by executing a NOP in its place.

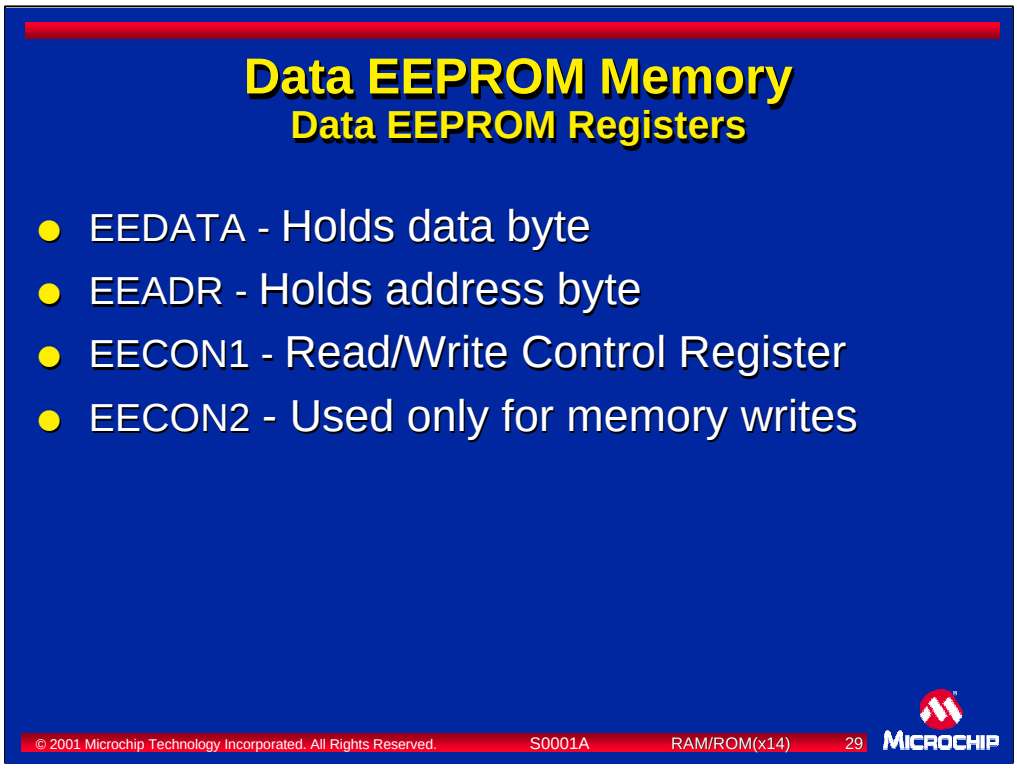
Using indirect addressing allows using a loop to perform repetitive tasks, and requires only 7 lines of code. If the same task were to be performed with direct addressing, approximately 95 lines of code would be required.



Data EEPROM Memory is the third memory type and provides non-volatile storage of data. Data EEPROM Memory can store up to 256 bytes and is accessed via the same special function registers that are used for reading and sometimes writing FLASH Program Memory.

It is offered in FLASH devices that offer writing to program memory using firmware. Check the data sheet for the amount and functionality of data EEPROM memory.

Data EEPROM memory has high endurance. In other words, it can be written to many times, generally more than 100,000 times. Program memory can generally be written up to 1000 times. Check your specific device datasheet for electrical specifications and programming specifications.

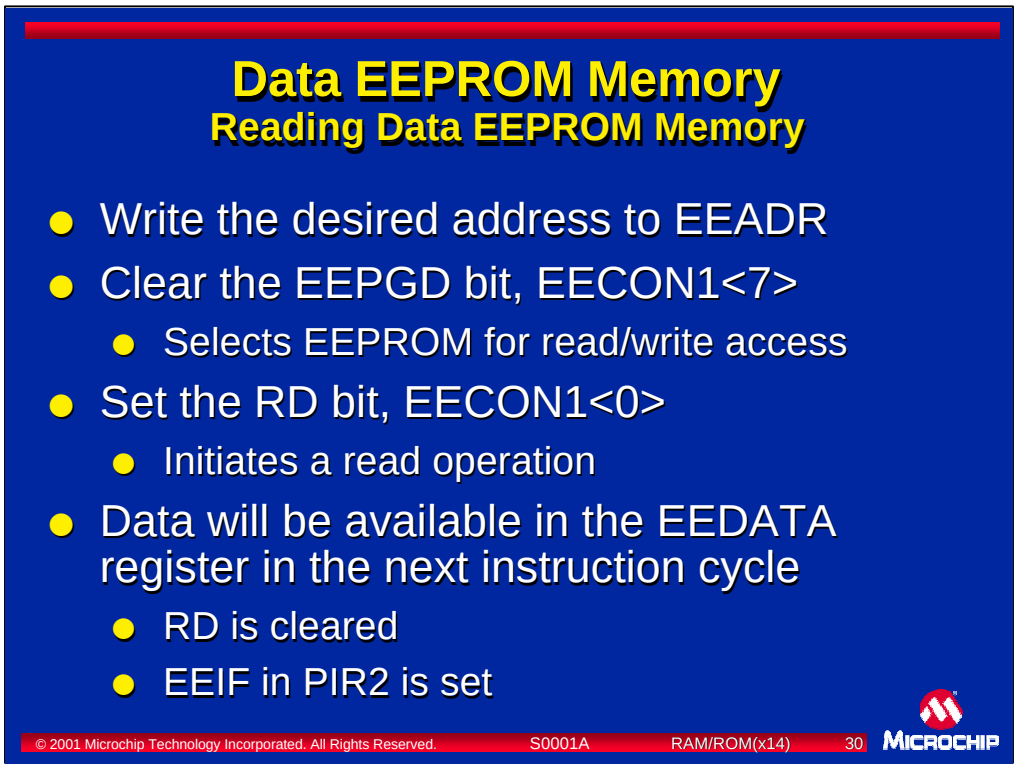


These are the registers that are used for accessing the Data EEPROM Memory. Their functions are the same as when reading or writing program memory.

The EEDATA register holds the 8-bit data value. The EEDATH register is not used.

The EEADR register holds the address which can be up to 8-bits depending on the size of data EEPROM memory. EEADRH is not used.

The EECON1 and EECON2 registers control the access as described for program memory reads and writes.

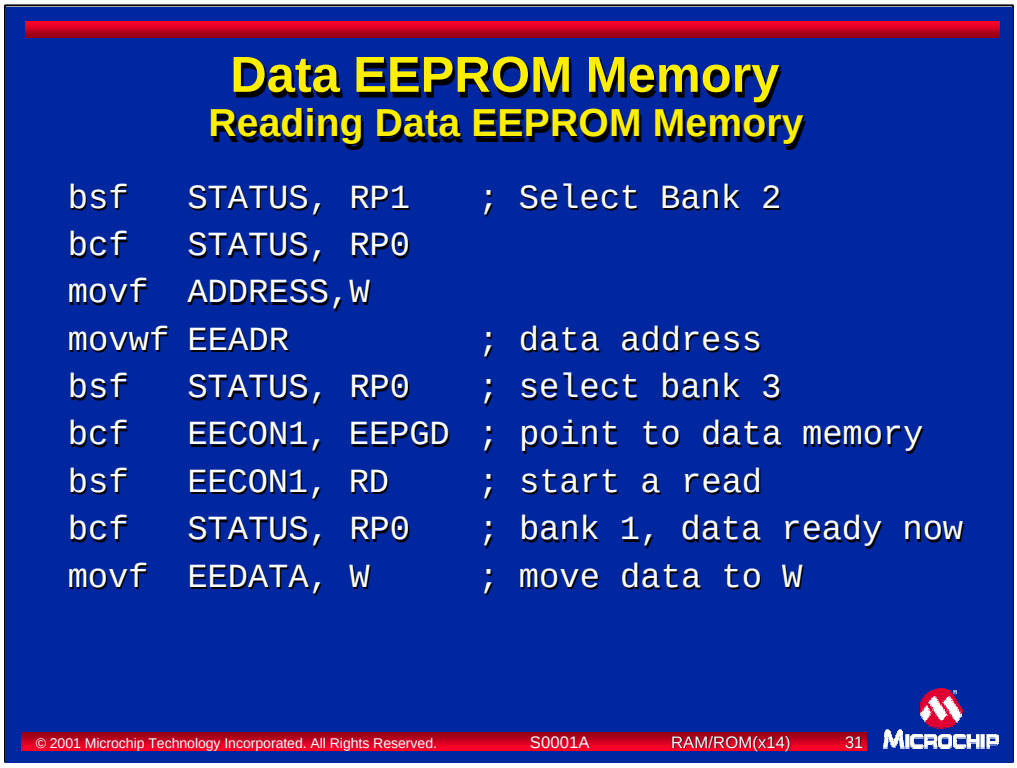


Reading Data EEPROM Memory is similar to reading program memory. The address of the desired memory location is written to EEADR. The EEDATH and EEADRH registers are not used.

The EEPGD bit in EECON1 is cleared to indicate the access will be to data EEPROM memory rather than program memory.

The RD bit in EECON1 is then set to begin the read. The data stored in the desired address will be available in the EEDATA register in the next instruction cycle.

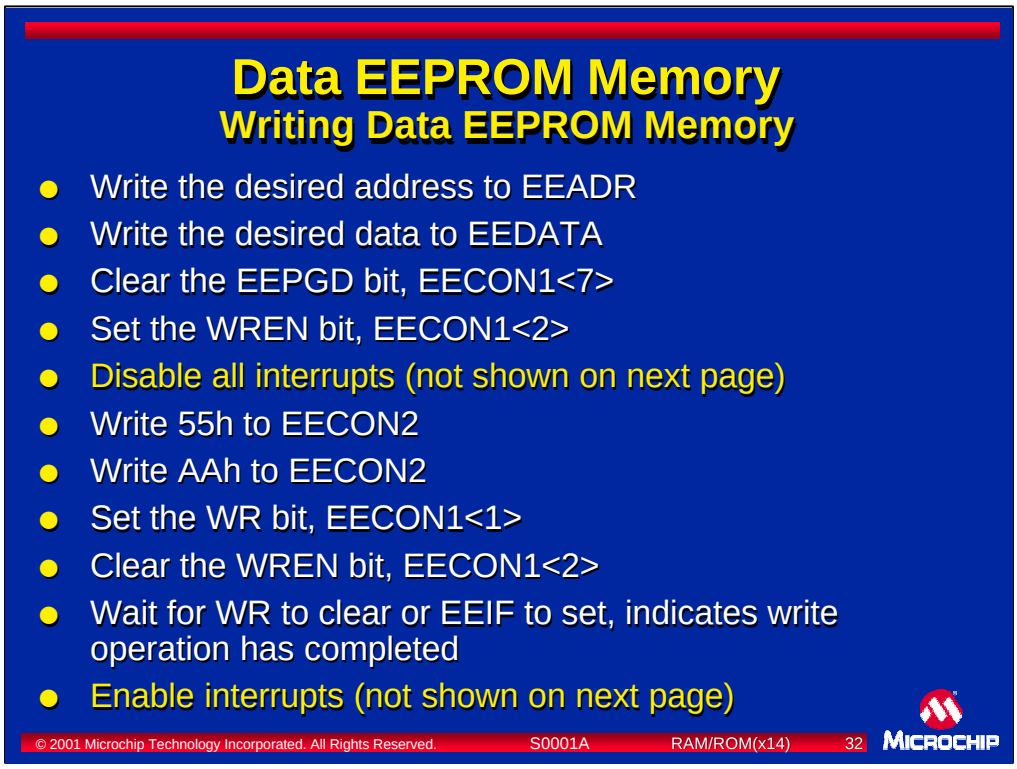
When the read is completed, the RD bit is cleared, and the EEIF bit in PIR2 is set, indicating an interrupt request has occurred.



This is a code example of reading a Data EEPROM Memory location. The address is stored in a RAM location called ADDRESS. The desired address is written to EEADR. The EEPGD bit is cleared and the RD bit is then set to start the read operation. When the read is completed, the RD bit will be cleared, and the EEIF bit in PIR2 will be set. Data will then be available in the next instruction cycle.

The differences between a data memory read and a program memory read are as follows:

1) Loading an 8-bit address for data memory, or 13-bit address for program memory
2) Clearing the EEPGD bit for data memory, or setting it for program memory.
3) When reading program memory, two NOPs must follow the instruction that sets the RD bit. They are not required for reading data memory

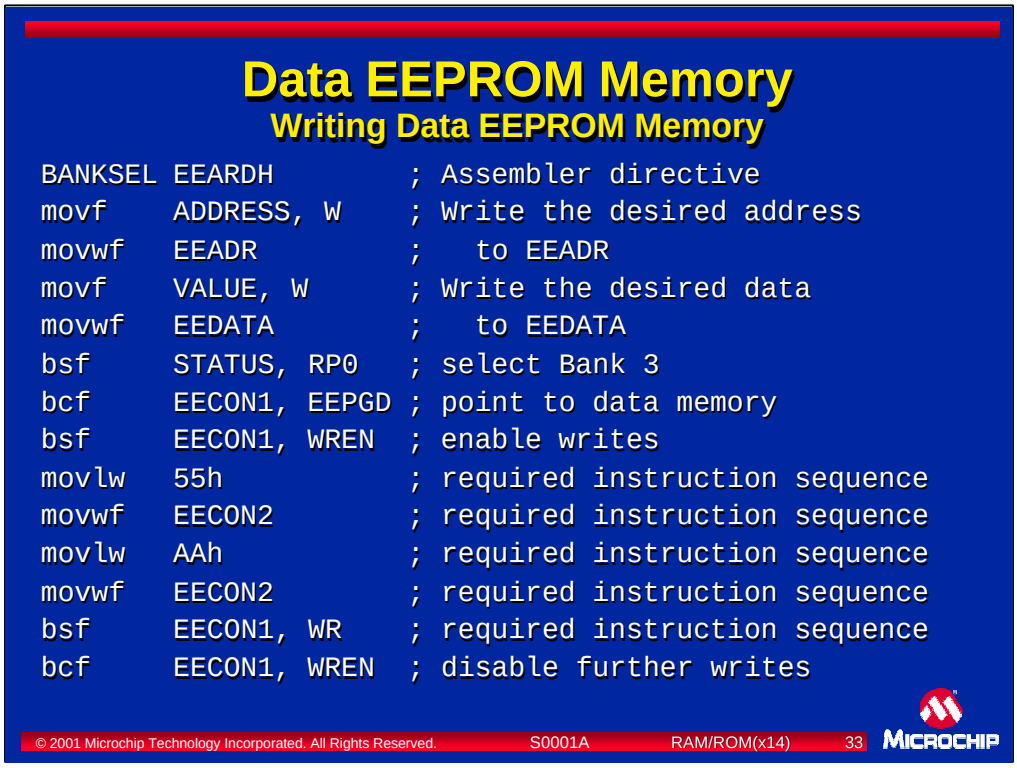


Writing to Data EEPROM Memory is very similar to that of program memory. The differences are:

1) Loading EEADRH is not required.
2) Loading EEDATH is not required
3) The EEPGD bit is cleared to access the data memory
4) Code execution continues during the write to data EEPROM memory
5) NOPs are not required after setting the WR bit.

The actual erase and write operations occur without affecting code execution. Two bits may be polled to determine if the write has completed. The write has completed if either:

1) The WR bit has been cleared, or
2) If the EEIF flag is set (this flag must be cleared before setting the WR bit). If interrupts are re-enabled after the WR bit is set, an interrupt can be generated when the write completes.



This is a code example of writing to a Data EEPROM memory location. The desired address is written to EEADR and is contained in the RAM location called ADDRESS. The data value to be written is loaded from the RAM location VALUE into EEDATA. The code clears EEPGD to select data memory and enables write operations by setting the WREN bit. The special 5 instruction sequence is then executed followed by clearing the WREN bit.

The differences between the data EEPROM write code and the program memory write code are

1) Writing the upper byte of the program address is not required
2) Writing the upper byte of the program word is not required
3) Setting the EEPGD bit to select program memory, or clearing it to select data memory.
4) Inserting two NOP instructions after the instruction that sets the WR bit is not required
5) Note that a program memory write operation will halt the execution of instructions while the data memory write operates in the background allowing instructions to continue executing.

Now that you've gotten started understanding the PICmicro MCUs "On-chip Memory" you'll want to learn more about Microchip Technology's synergistic product portfolio. A good place to start is the engineering website at www.microchip.com