Breaking Through the 1 MByte Address Barrier Using the Am186ES Microcontroller

The x86 architecture has come to dominate the microprocessor landscape as the most successful architecture in the world. The 186 is the 16-bit microcontroller version of the x86 architecture and it has had similar success in the 16-bit embedded market. New versions of the popular 186 have provided more performance, such as the Am186ES microcontroller that provides 5.35 Dhrystone 2.1 MIPS using 70 ns memory. One area that still provides a barrier to some applications is the 1 MByte linear address range of the 186. This is the same 1 MByte address range of the original IBM PC. Breaking this 1 MByte address range is not as easy as setting a compiler switch, but it can be done with a little software planning. Using the programmable I/O and the unified middle chip select capability of the Am186ES microcontroller, this greater than 1 MByte address range can be implemented directly to the memories with no external hardware. The SRAM interface timing, and byte write enables among other system functions are integrated into the Am186ES microcontroller.

Consider the simple case of a typical system with Flash and SRAM that needs additional data space beyond the 1 MByte boundary. Examples of what this data may be include maps, tables of information or any data base that is accessed in blocks, rather than randomly throughout the data base. In the case of the map, all the data associated with a grid would be placed in one segment. A database example may be all the sales made on a certain day.

Figure 1 shows the hardware for an application requiring more than 1 MByte of memory. The upper chip select (UCS) accesses the Am29F010 Flash containing 256 KBytes of application instructions, interrupt service routines and data that must remain available for subroutines (for example, text strings for output). The lower chip select (LCS) accesses to provide the interrupt vector table, stack space and any other volatile memory requirements. The middle chip select 0 (MCS0) connects to the additional Am29F016 Flash memory and is set for 256 KBytes. The unified middle chip select capability of the Am186ES means that larger sections of memory can be accessed through the single chip select. The programmable IO (PIO) act as additional address bits into the Am29F016 Flash memory. In this configuration, there are 256 KBytes of SRAM, 256 KBytes of Flash in the Am29F010 always available, 256 KBytes of address space for other system requirements and 16 independent Flash segments of 256 KBytes each in the Am29F016 Flash. The total memory, SRAM and Flash, available is 4.5 MBytes. Obviously, there is still a great deal of flexibility remaining in the address map to adapt this to a particular embedded application’s requirements. Larger or smaller memories could be used as needed.

Figure 1. The Am186ES with extended Flash Memory

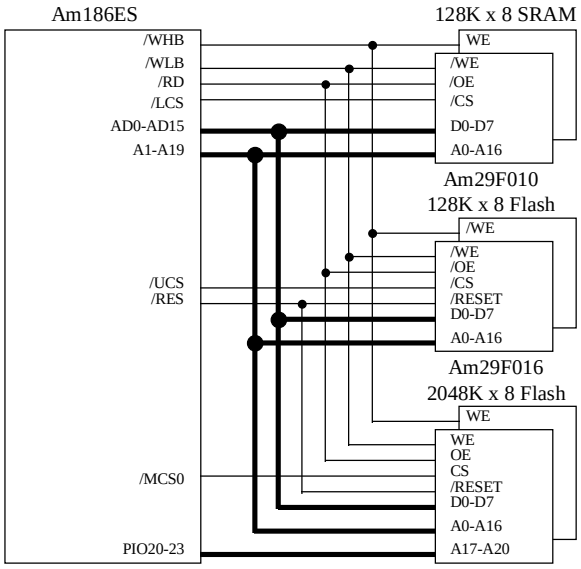
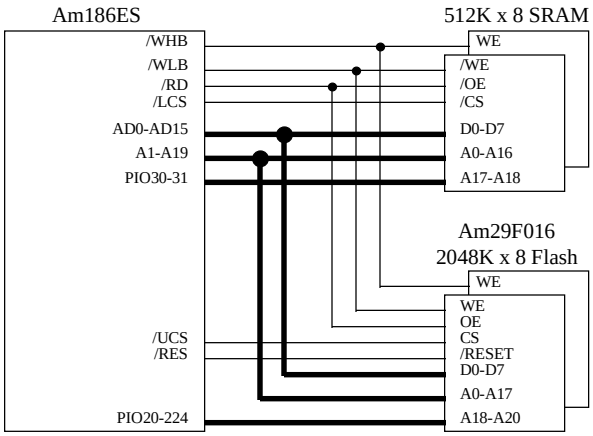


Figure 2. The Am186ES with extended single banked Flash and SRAM memory



Whenever a new segment of memory is needed from the Am29F016 Flash memory, a software routine changes the PIO for the new segment address. If an interrupt service routine needs to access data in the Am29F016 Flash memory, then four steps occur. The interrupt service routine first saves the current state of the PIOs and then changes the PIOs to use the new segment. Next, the interrupt service routine executes whatever tasks it needs to do and then the interrupt service routine restores the PIOs before exiting. Using the example of the map, the software would first decide what map grid was to be displayed, change the PIOs to correspond to the part of Am29F016 Flash memory that has the data for that map grid, then let the display subroutines access the data in the Am29F016 Flash memory. When the user changes to another grid, the higher level software would again match the PIOs to the new map grid data. If the user asked for more detail about a feature on the map, an interrupt service routine would intervene. The interrupt service routine could save the PIOs associated with the current map grid, change the PIOs to the Am29F016 Flash memory that had the additional detail information and the return the PIOs to the map grid address when the user was finished with the additional detail information.

If the application needed additional address space for instructions, a similar approach could be used. The hardware remains the same as figure 1. The subroutines used commonly throughout the application (for example serial port drivers), would reside in the Am29F016 Flash accessed by the UCS. This technique will work best if the software has distinct modes. An example of this would be a personal digital assistant. Though some of the subroutines would be common (for example keypad interface), each embedded program (for example word processing, spreadsheet, etc.) would have its own distinct code. Another example would be an application with code that is only used once (for example start up code) or interrupt service routines that are separate from the main application. The Am29F016 Flash memory could be used for embedded programs, code only used once or interrupt service routines. Before calling the embedded program or interrupt service routine, the PIOs are set to address the new code segment. Each interrupt service routine would have just enough code in the Am29F016 Flash to save the current PIO setting, change to the new PIO setting and to restore the original PIO setting when the main portion of the interrupt service routine returned from operating out of the

Am29F016 Flash memory. When linking the program, each 256 KBytes segment in the Am29F016 Flash is linked separately. This allows for the linker to resolve subroutine accesses made from each separate segment of the Am29F016 Flash with the commonly shared code in the Am29F010 Flash.

The key to using both the expanded data and expanded code address space is to partition the data or code from the top down. At the highest level, the programmer must think about the code or data and its interaction with other parts of the program. This is why subroutines that are executed only once (for example, code used for start up initialization) and interrupt service routines are prime candidates for dividing into the additional segments. The nature of such code lends itself to the high level division.

If the user wants to have a single bank of Flash or single bank of SRAM, the same idea can be used for breaking the 1 MByte address boundary. However, some additional care must now be taken with the interrupt service routines. Figure 2 shows the implementation of extended SRAM and Flash. LCS is the chip select to 1 MByte of SRAM. Two PIO to the upper address of the SRAM splits the SRAM into four segments of 256K each. UCS is the chip select to 4 MBytes of Am29F016 Flash. Four PIOs separate it into 8 segments of 512 KBytes each. The remaining 256 KBytes of address space is available for peripherals specific to the embedded application.

On boot up, the integrated pull-ups of the Am186ES microcontroller PIOs let us access the 512 KBytes segment of the Am29F016 containing the start up code. If the Am186ES microcontroller did not have the integrated pull-ups, then external pull-ups would be required. Otherwise, the upper four address bits of the flash would float and you could not reliably know which of the 8 segments of the Am29F016 was being accessed.

As in the examples of figure 1 and figure 2, the software switches the PIOs to change segments as required. What has changed is that you can not know which SRAM or Flash segments the program will be in when an interrupt occurs. The Am186ES microcontroller will always get the interrupt vector from the interrupt vector table starting at address 0. This means the bottom 1 KByte of each segment of SRAM must have a copy of the interrupt vector table. By the same token, you can not know which of the 8 512 KBytes segments of Flash the program will be in when the interrupt occurs. There are two ways to handle this. One is to duplicate enough of the interrupt service routine in each 512 Kbytes segment of Flash to change the PIOs to the correct 512 KBytes segment of Flash and 256 KBytes of SRAM. The other method is to use the SRAM to contain the code needed to switch to the correct Flash and SRAM segments.

One side affect of this technique for breaking the 1 MByte boundary is the inherent protection of the expanded segments. No segment can address another segment without first changing the PIOs. The stacks for interrupts and main program can be maintained separately, lowering the chance that the maximum stack expansion for main program and interrupt service routine could together overflow the allotted stack space.

Using these techniques greatly extends the range of 186 applications. The improved performance of the more up to date implementations of the 186, like the Am186ES microcontroller, provides more than enough horsepower to implement the additional code to switch the PIOs. All or part of these methods can be implemented to meet the specific needs of your embedded application.