FUJITSU

Application Note

External EPROM Power Down for the

MB89630 Micro controller

Vers. 2.0 by E. Bendels

This Application Note shows a concept on how to realize a low-power consumption system even when using an EPROM connected to the external program bus.

The problem:

Usually, theere is a big difference in power consumption between a single-chip microcontroller system, and ROMless microcontroller systems.
A single-chip system requires just one chip, where as a multiple-chip system using a ROMless controller will require at least an additional program memory (e.g. ROM, EPROM or FLASH) and an additional address latch (e.g. 74HC573). Of course, more chips consume more power.

But, the main reason why a single-chip controller can dramatically reduce its current consumption is because the controller usually does not have to run all the time.
The single chip-controller can go into a power-saving mode where some of its components like the internal ROM are simply powered-off.
When the controller wakes up again, for example due to an internal timer interrupt, the controller hardware makes sure, the internal ROM is powered-on again before program execution continues.

A typical average current consumption value in this cases could be lets say 10 µA, even if the controller will draw lets say 5 mA while it is running for short periods.
(Of course exact figures depend on the supply voltage, main-clock speed etc.)

In contrast, for the multiple-chip case, the external EPROM is the bad guy.
It will draw 100 µA to 1000 µA alone, even when its not needed and the controller is in the power down phase.

The basic solution

To solve the high standby current problem of the EPROM in multiple-chip designs, we could apply basically the same scheme as the single-chip controller does, by simply putting a power switch transistor in the power supply lines for the external components. See block diagram in fig. 1.
Thus, while the controller is in a power-down mode and does not need to access the external memory, it could simply be powered-off.

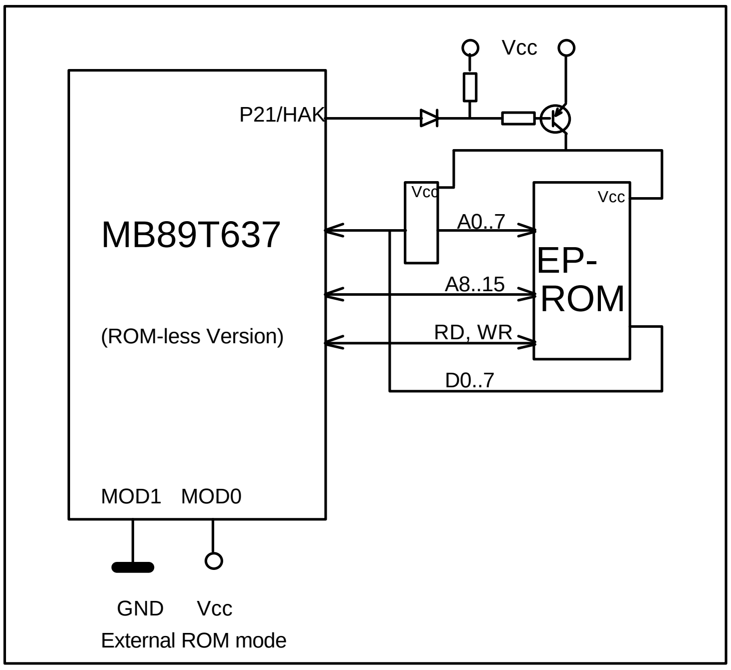

Fig.1, External EPROM Power Control

Unfortunately its not as simple as that, as the following considerations will show.

Detailed solutions for ROMless systems

Problem A: The chicken and egg situation

Lets assume the controller program has powered down the EPROM and activated a power down mode lets say the watch mode.
After some time (lets say one second which is a rather long time for a controller), the watch timer will wake up the controller.
In the single-chip case, some special controller hardware will power on the internal ROM first, before program execution continues.
In our case, we would have to power-on the external EPROM by software which controls the transistor via an I/O pin, but where should the software come from when the EPROM is not powered-on yet.
(Note: A similar problem exists for powering-down and activating the watch mode.)

Solution A: Instruction Code in the internal RAM

A solution for providing program code, even if the external EPROM is powered off is quite simple. The F^2MC8L controllers can execute its program from within the whole address range, also the internal RAM. Thus, we can install some program code in the internal RAM area by copying it from the external EPROM during processor initialization after reset, when everything has to be active.
To power down the system, the controller will jump into the RAM code, switch off the external EPROM and execute a power down instruction.
Later on, at wake-up time, the controller program should continue from this internal RAM and activate the external EPROM before continuing program execution from this memory.

Problem B: The controller does not want to continue program execution from internal RAM.

Experiment users of our controllers might have seen this problem already:
The controller ´wake-up´ is done via an interrupt, for example the watch timer interrupt who could wake up the controller after 1 second of sleep. What the controller usually does when it receives an interrupt is to read the appropriate interrupt vector and execute an interrupt service routine (ISR). Unfortunately these interrupt vectors are in the ext. EPROM which is not powered-on yet.

Solution B: Interrupt wakes up the controller, but interrupt vector call is disabled

To overcome this problem, we do something unusual: We will disable all interrupts using the CLRI instruction, before activating the power down mode.
If an interrupt occurs now, the controller will still wake up, but it will continue program execution right after the power-down instruction, which is inside the internal RAM.
Here we can activate the external EPROM and then enable all interrupts again using the SETI instruction. And now the interrupt request will be served by the appropriate ISR as usual.

Problem C: How to start the system after Reset/Power-on

So far, things look quite good, but we have to take care about how to initially power-on and off the external EPROM and probably other external devices.
Since the EPROM is controlled via a controller port, we have to make sure the EPROM is powered on during and after a RESET, so the controller can execute some initialization code.

Solution C: The right I/O pin:

It just happens, that there is one port signal which does exactly what we need.
The P21-/HAC signal is a general purpose output port forced to low-level during and after RESET. This is the ideal signal level to simply drive a simple PNP or enhanced mosfet transistor switching to power power supply for the external EPROM and other peripherals like the address latch.

Problem D: Leakage trough EPROM protection circuitry

The solution seems to work so far, but we do not have to forget another important detail.
Controlling just the Vcc line of the EPROM and address-latch to power-off these devices is not enough.
We have to make sure that none of the other connections (address, data and control signals) are at voltage levels higher then ground.
If that was the case, the EPROM would be powered net via the Vcc line, but through its input protection circuitry which becomes active if any input voltage is higher than the Vcc level.

Solution D: Tristate the external bus interface:

Fortunately the controller has got a control bit, which can specify that the output signals of the external bus interface shall go into high-impedance once the watch or stop mode is activated.
(Bit 5 in the STBC-register).
Note that in this case also P21 becomes tristate, so a pull-up should be provided to make sure the transistor keeps open during the power down phase.

Problem E: Short Leakage currents trough protection circuitry during wake-up process

Assuming the controller was powered down and is waking-up now.
The first thing which happens is, that the controller hardware will switch its external bus from tri-state to active state again.
Then program execution (as we described above) will continue from internal RAM, and the software will finally power-on the external EPROM again.
For the time in-between, there was that undesirable situation, that a current went over the bus signals into the EPROM as described in problem D.

Solution E: Let the controller hardware control P21

The simplest solution to this problem is to leave P21 at low-level and not to switch it by software.
The controller will do this automatically when activating the watch-mode, when it tri-states the bus-signals and activates it again immediately at wake-up.

The only remaining drawback with this is, that when implementing longer power-down phases, e.g. wake up happens every 1 second, but the software in RAM will immediately activate the watch mode again and will call more complex routines in EPROM only every 10 seconds,
the EPROM is also automatically shortly powered on every second.
A final solution to overcome this would be to insert special gates/switches into all bus-signals and go back to the P21 switching by software.

The example system

An example system was developed on a private basis to actually verify the above descriptions. This test-system contains even some more external components so the board is of more universal use. See the block diagram in Fig. 2.

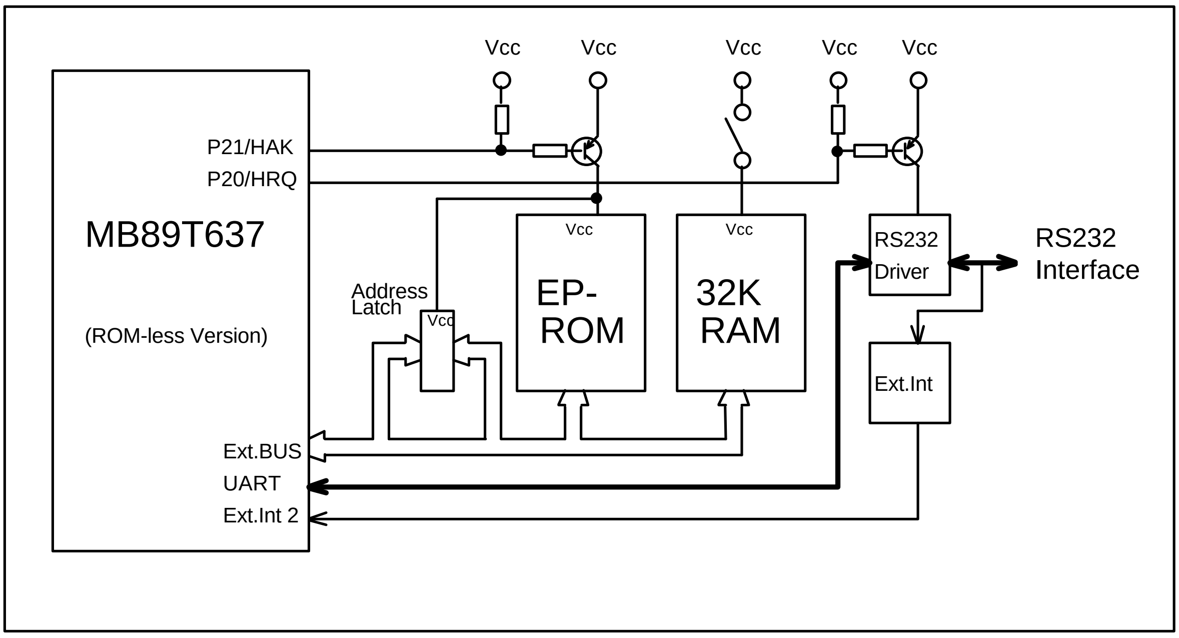

Fig. 2, Test System

Basically an optional external SRAM can be added, and a serial RS232 interface, where the driver IC can also be powered on or off.

Some Test results:

Using the watch-mode, average current figures of about 20 µA can be achieved.
In case the stop-mode can be used (controller does not have to wake up itself), the consumption goes even down to about 1 µA.
(Note: In this case activity on the RS232 line could wake up the controller via the Ext.Int. input.)