The TI MSP430 microcontroller

Introduction

The Texas Instruments (TI) MSP430 family of processors are low power 16 bit devices. They are marketed at low power applications such as battery devices, however in spite of this they contain a sophisticated processor core and depending upon the model, a useful array of integrated peripherals. TI produces a set of microcontrollers based around a core 16 bit CPU as shown in Figure 1. Different members of the family include different peripherals and different amounts of memory. They all include a JTAG interface. (JTAG stands for Joint Test Action Group and in this context represents a standardized interface to the microcontroller that can be used for control and monitoring during program or system testing and debugging). Product designers choose the particular family member that best suits their application, generally speaking, cost and power consumption increases with sophistication.

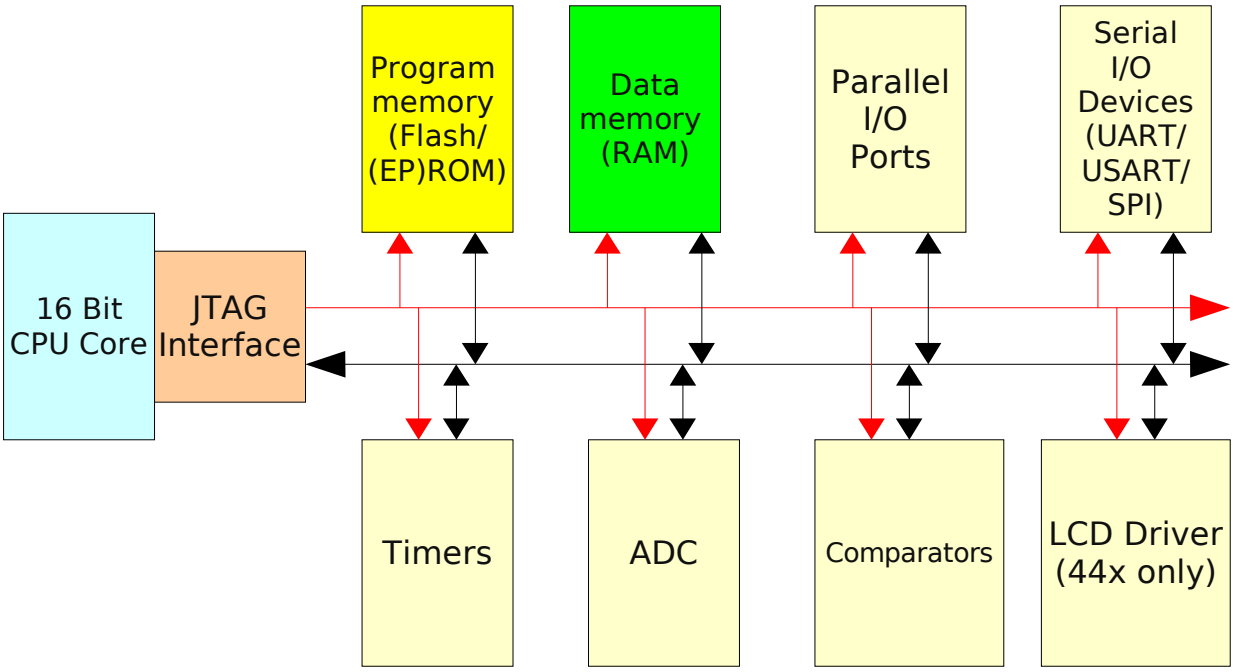


Figure 1. The TI MSP430 family

The CPU Core

The CPU consists of an instruction decoder, arithmetic logic unit, and a register file (group of registers). The instruction decoder is responsible for translating the numeric program instructions into processor actions, the arithmetic logic unit carries out additions, subtractions, logical operations and so on. The register file consists of 16, 16 bit registers (a register is simply a storage location inside the processor that can hold a single 16 bit number). The registers are numbered R0 to R15. The first 4 of these (R0 to R3) have special designation, the remainder, R4 to R15 are for general user e and may be used for example as instruction operands.

R0 – The program counter (PC)

At any given time, the program counter contains the address of the next instruction to be executed. It always contains an address that is a multiple of 2, this is because memory is addressed in 2-byte (16 bit) chunks by this CPU. When executing a linear sequence of instructions, the PC is automatically moved on to the next instruction by the instruction decoding logic. Programs can also write to PC to effect a jump to a particular location.

R1 – The Stack pointer (SP).

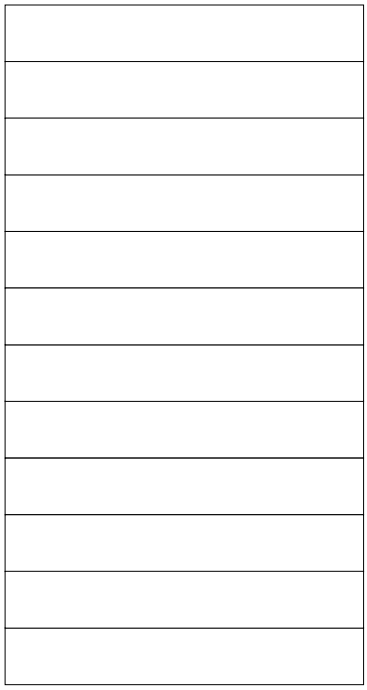
Foreword: Stacks and microprocessors.
Imagine you are an over-worked employee in the company who is continually being given new tasks to perform. Now, suppose you are in the middle of some complex job and your employer requires to temporarily divert you to some other job. Being used to disturbances like this, you decide to write in your notebook the current status of the job you are working on before you move on, thus allowing you to take up from where you left off when the other job is done. Now, suppose that while you are doing this new job, you are disturbed with yet another more urgent task. Again the notebook comes out… The function of the notebook in the above is to store the current status of each job for retrieval later on. In effect, the notebook is like a memory device used for storing state information. A microprocessor stack performs a very similar function. The stack in a microprocessor is an area of memory set aside for storing the microprocessor state (contents of registers and so on) in the event of a disturbance. Suppose the microprocessor is doing a job (1) below and is disturbed (the exact mechanism for this will be shown later). If it wishes to return to this job at a later time, it must store its current state in the stack as shown below (2).

(1)

Microprocessor performing Task 1

Working Registers

The Stack

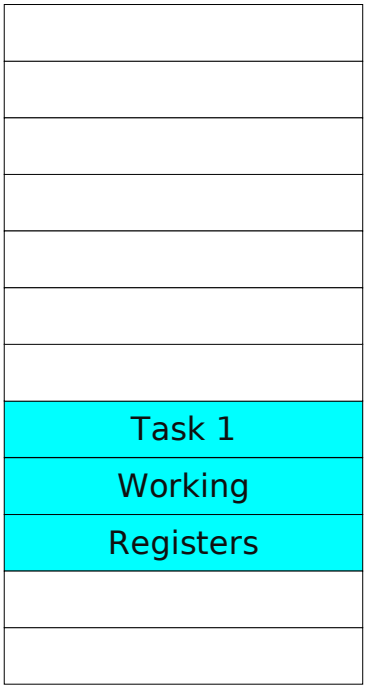


(2)

Microprocessor performing Task 2

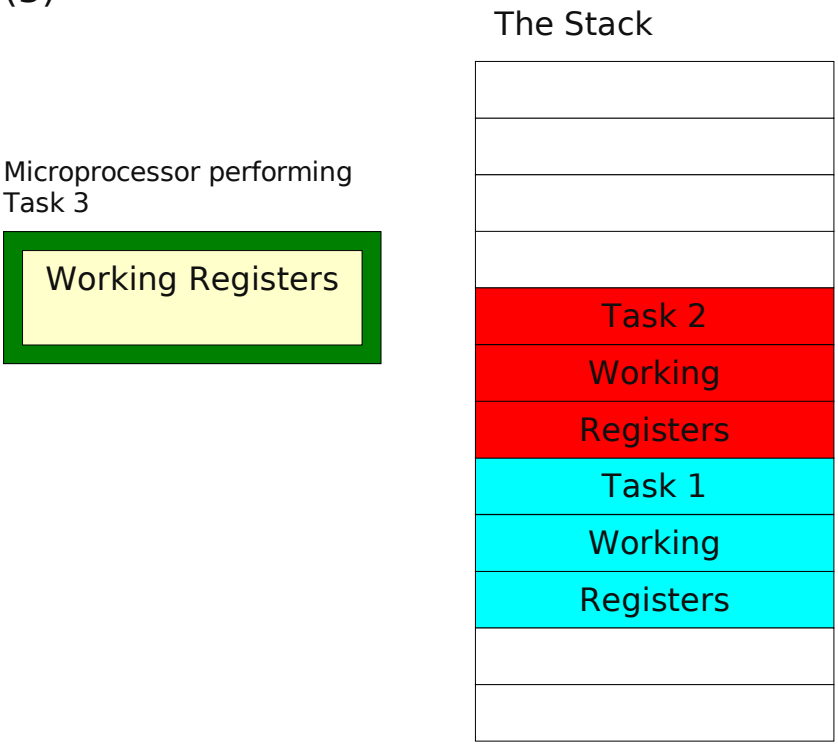
Working Registers

The Stack



Now, suppose Task2 is disturbed, again, the microprocessor state must be saved to the stack as shown below (3).

(3)



As you can see, the register contents for Task2 are stacked on top of Task1 (hence the term 'Stack'). Now suppose Task3 is completed, and the microprocessor wished to return to its previous job. It simply pulls from the stack the register contents for the job that was interrupted last and carries on (back to (2)). When this job is complete, it again returns to the previous state by pulling from the stack the previous state of its registers.

Some important points should be noted here:
The stack is a Last in, First Out mechanism.
The stack resides in memory, just like the program and the data so care must be taken not to upset its contents lest the microprocessor lose track of what it is doing.

Implementation of the stack on the TIMSP430

The MSP430 uses R1 as a pointer to the stack in RAM (the stack must reside in RAM). At any given time, the stack pointer points to the last value pushed (placed) on the stack. Values are pushed as 16 bit words, and after each push, the stack pointer is decremented by 2. As values are taken (pulled) from the stack, the stack pointer is increased by 2. The stack therefore operates in an inverted fashion. This approach is common among processors, the reason being that it allows the stack to begin at the top of memory and programs to begin at the bottom of memory.

The MSP430 instruction set includes 4 instructions which affect the stack. These are:
PUSH *operand* – SP is decremented by 2 and the operand is copied to where SP points
POP *operand* – The value that SP points to is placed in the operand, SP is incremented by 2
CALL *subroutine(address)* – SP is decremented by 2, PC is copied to where SP points, subroutine
address is placed in PC.
RET – The value that SP points to is placed in PC, SP is incremented by 2.

R2 – The status register (SR)
This register is used to flag significant calculation results such as overflows and to control the operation of the processor. (R2 also forms part of the constant generator system of the processor will be mentioned in the next section. Suffice it to say for now that R2 can behave in different manners depending on how it is used in an instruction. If it is treated as a 16 bit register is behaves as the Status register.)
The SR should be viewed as a collection of individual bits, each of which has a particular meaning. The bits are laid out as show in Figure 2:

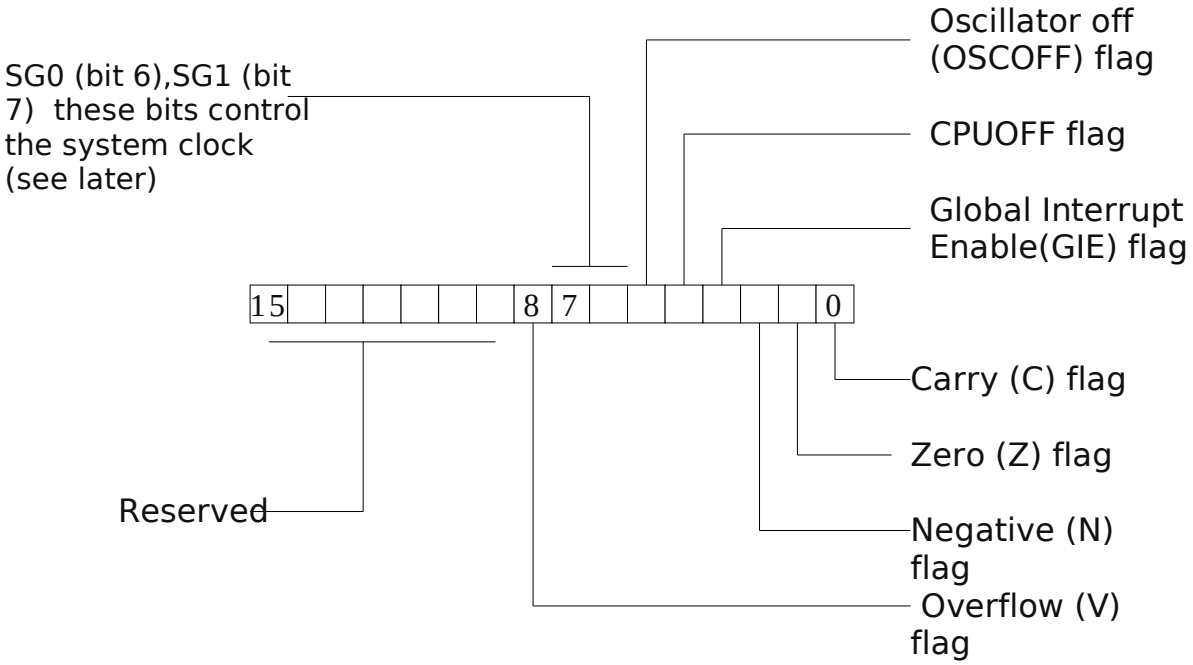


Figure 2. Layout of bits in the SR register

Using the arithmetic flags.

Microprocessors have the ability to make decisions. These decisions normally take the form:

If a particular condition is met then execute one part of the program

otherwise

Execute a different part of the program

The MSP430 instruction set includes a group of instructions designed specifically for making decisions such as this. These instructions are the Conditional Jump Instructions. A conditional jump instruction operates examining by the Status Register (SR) and, based on its contents will change the contents of the program counter (PC) thus changing the course of execution of the program. The Status Register contents are changed each time the ALU performs a calculation. For example, suppose the sum 3-2 is performed in the ALU. The result is obviously negative, and, the negative flag (N) is set (or “raised”) in the Status Register. Conditional jump instructions such as JN (jump if negative flag is set) can be used to execute instructions in response to this condition. The four arithmetic flags are now discussed in detail.

The N flag.

Consider the following code fragment:

```
        MOV  #12,R4
        SUB   #13,R4
```

The first instruction places the number 12 in the R4 register (overwriting what was previously there). The next instruction subtracts 13 from R4. The calculation can be written as follows:

```
        0000 0000 0000 1100        (12 decimal)
-       0000 0000 0000 1101
=========================
        1111 1111 1111 1111
```

This is equivalent to -1 in 16 bit binary terms (as a general note, if the MSB is 1, then the number may be considered to be negative). The N flag is set following this calculation.

The Z flag.
This flag is raised if the result of the last calculation was zero.

V: The oVerflow flag.

This flag is important if you are performing calculations with signed numbers. Consider the following calculation:

```
        0111 1111 1111 1111 (hex 7fff or decimal 32767)
+       0000 0000 0000 0001
        ================
        1000 0000 0000 0000 (hex 8000 or decimal +/- 32768)
```

The MSB in this case is a 1 which, depending on context and your application may indicate that this is a negative number. This makes no sense however as the addition of two positive numbers can never produce a (correct) negative result. The V flag is set in this case as a warning of the possible misinterpretation of the sign of this number. There are a number of other calculations that may produce such errors – can you think of any?

The Carry flag.
The carry flag is used to indicate that a result is too big to fit into the target register. Consider the following:

```
        1111 1111 1111 1110
+       0000 0000 0000 1110
        ================
        0000 0000 0000 1100  and a '1' carried out.
```

The result of this addition is too big for a 16 bit register and the carry flag is set to indicate this. Programmers can make use of the carry flag to write programs that can add arbitrarily large numbers.

R2/R3 Constant generators.
The MSP430 has an unusual feature in that it has register support for 6 constant numbers that occur frequently when writing programs. It might seem bizarre that support for such a meagre number of constants was included however, given the frequency with which they occur, the potential performance gain in the form of less traffic on the system buses was considered worth it. For a thorough discussion of the constant generator subsystem and its (transparent) use by assemblers to emulate instructions, see the MSP430 user's guide.

R4-R15 General purpose registers.
These registers may be used as source or destination operands for most of the MSP430 instructions. The presence of such a large number of general purpose registers allows programmers to minimize memory access and code size (as variables may remain resident in registers). The default word size for the MSP430 is 16 bits however 8 bit data movement/manipulation is possible by appending ".B" to the relevant instructions.

Simple programming examples

Addition of two constants.
Suppose we want our microcontroller/microprocessor to perform the following action: Add the numbers 6 and 9 and store the result in memory location 200. The assembler code to do this is shown below:

```
mov #6,R4              ; place the number 6 in register R4
add #9,R4              ; add 9 to it
mov R4,0200h           ; store the result in 200h
```

Before this program can be run on the msp430 it must be converted to a sequence of numeric instructions (machine code) using an assembler. Now suppose that the assembler (for reasons we shall see later) places the machine code in memory starting at address 110E. A memory image of this area is shown below:

Address	Contents (16 bit words)	
110E	4034	
1110	0006	These two words constitute "mov #6,R4"
1112	5034	
1114	0009	add #9,R4
1116	4480	
1118	F0E8	mov R4,0200h

From the above, we can conclude that instruction number 4034 means, put the 16 bit number that follows into Register R4. Similarly instruction 5034 means add the 16 bit number that follows to R4. The third instruction is a little more interesting. Instruction number 4480 means store the contents of R4 at the address which is F0E8 bytes away from the current location (note current

location here means the location of the number f0e8). Adding F0E8 and 1118 we get 10200 – a 17 bit number. The msp430 is however a 16 bit device so the leading '1' is dropped leaving an absolute address of 0200 – which is where we wanted to put it. In this case, the compiler figured out relative distance between the instruction and the intended target.

In the previous example the "#" (hash) symbol was used to indicate a literal value i.e. The actual number 6 or the actual number 9. If this symbol is left out, the compiler assumes that the operator refers to a memory location i.e. it signifies the address of something. The next example illustrates this.

```
mov 0202h,R4      ; load the contents of memory 202 (and 203)
add 0204h,R4      ; add the contents of memory address 204 (and 205)
mov R4,0206h      ; store the contents of R4 at memory address 206,207
```

Encoded as :

Address	Contents (16 bit words)	
110E	4014	
1110	F0F2	equivalent to: mov 0202h,R4
1112	5014	
1114	F0F0	add 0204h,R4
1116	4480	
1118	F0EE	mov R4,0206h

Note the use of relative addressing for all operands.

In the past, assembly language programming was often used to produce efficient (fast) code for time critical applications. Those days are just about gone now. Modern processors are not as straightforward as their predecessors. In many cases, processors execute instructions out of order in an effort to make best use of their internal structures and system bandwidth. Speculative execution (the processor executes both branches of an "if" condition for example to keep its instruction pipeline full) is also quite common these days. Given the non-linear sequence of program execution the probability of human "hand-crafted" code being more efficient than compiler produced code is now quite small. For this reason code examples in the remainder of these notes will be predominantly in C.