RISING TO ZERO

We have now had a fairly thorough look at the addressing modes available on the 6809 processor, particularly the use of indirect addressing. There are still some variations we will need to discuss in more detail in the course, most notably the use of the program counter in indexing. For the moment, let's take a closer look at how the stacks are used.

So far in the course, we have used the two stack pointer registers, S and U, only as extra index registers. The use of the so-called 'hardware stack' for the storage of return addresses on subroutine calls has also been mentioned, although only in passing. Now we need to backtrack a little and consider the architecture of a stack, and the way it is used.

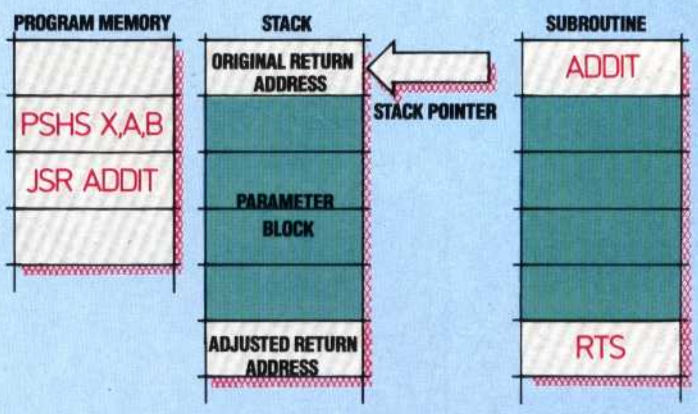
A stack is a special instance of a more general type of data arrangement known as a *list*. You should be familiar with the everyday idea of a list, even if you know little about the increasingly popular list processing languages, such as LISP and LOGO. A list is simply a sequence of data items. This sequence can be arranged in an order determined by some property of the data (for example, a series of numbers in numerical order, or a string of characters in alphabetical order), or it can be a random arrangement determined by the order in which data items were added to the list. With all of these lists it is sensible to attach significance to the identity or value of the 'next' or the 'previous' item in the list, and particularly to the list's first item (known as its 'head') and its last item (the 'tail').

One important feature of a list is that it is a *dynamic data structure*; that is to say, items of data can be added to, or taken from, the list at will. In a general list, data can be added or removed at any position in the list. The particular restriction that specifies that a given list is a *stack* is that data can be added to, or taken from, a stack only at one end. Each new item added to a stack becomes the 'listhead', and only this can be removed from it.

The name itself gives a good idea of the way a stack operates. Consider a stack of plates in a canteen: as a plate is needed it is taken from the top, and clean plates are put only on the top of the stack. You could add plates to, or take them from, the middle of the stack, but this would be unnecessarily problematic. It is possible, however, to inspect an item anywhere in the stack.

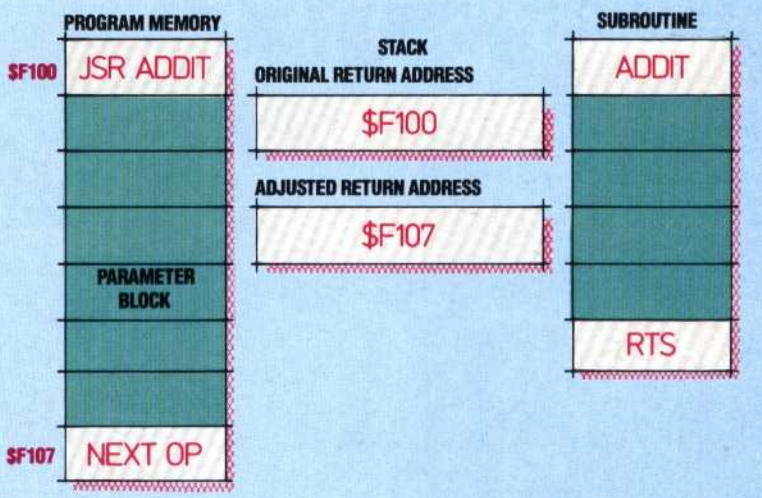
There are two extreme situations that can arise when a stack is operating: either the stack becomes empty, which is no problem if the next stack operation adds an item to it, but could be awkward otherwise; or alternatively, the stack could fill to overflowing. This second situation can be better visualised if we consider our stack of plates in a canteen: there would come a point where the stack

Parameter Push



Parameters can be passed to a subroutine by loading them into registers and then pushing them onto the stack. The subroutine can pull them off the stack, taking care to move the JSR return address down the stack when the parameters have been accessed. If this is not done, then the stack will grow continually, and eventually overflow

Parameter Insertion



A more usable method of passing parameters is to insert them into the program directly after the JSR call to the handling subroutine. The subroutine can then use the return address on the stack as the base address of the parameter block, and access it by indexed addressing. The return address must then be adjusted to point to the next program instruction, rather than to the start of the parameter block