circ\_FIR1: // Normal circle

Unlike the non-circular algorithm, a circular buffer works by changing which position in the buffer each new piece of data is written to, rather than writing to the first position and shuffling all the data up one space. This means we need a variable which keeps track of which position the newest piece of data was written to – in our program, this is the newest variable. newest is initialised to the last index of the buffer, and is decremented to the previous position each time data is read in, meaning data is stored in descending order of array index. When newest reaches the start of the array, it is explicitly set to the last position again, to prevent underflow.

The reason we need this variable is to keep the data buffer aligned with the filter coefficient buffer, to make sure we are multiplying the correct pairs of values. To achieve this, the convolution loop initialises the data buffer index, j, to the index of the newest piece of data, and the filter buffer index, i, to 0. The convolution continues similarly to the non-circular algorithm, with both indices incrementing, except that since j started mid-way through the array, at some point it will overflow, so an if statement is added to check for this overflow and correct it by returning j to the start of the array.

circ\_FIR2: // Split loop

The condition in the if statement in the previous algorithm only ever evaluates to true once for the duration of the for loop, however it is run every loop, causing unnecessary extra cycles to be used. We can remove it by pre-calculating the point at which the overflow will occur, and splitting the function into two for loops instead. This way, the same operations are executed, without the overhead of the if statement.

circ\_FIR3: // Double size buffer

Another method of removing the if statement is using a data buffer of twice the length, and storing each sample at the index pointed to by newest, but also at the index newest + BUFSIZE. This way, we can start our convolution iteration over BUFSIZE elements without having to worry about overflowing the array, and also being confident that each of the data values is accounted for, as any stretch of BUFSIZE elements starting at any point will contain all the data samples. If we start at newest, then we can be sure the array is aligned properly also.

circ\_FIR4: // symmetrical circular buffer

A different approach is to reduce the number of operations by taking advantage of the fact that our filter is of linear phase and so its coefficients are symmetrical. Since the first/last coefficients are equal, as well as the second/second last and so on, we only need to perform the half as many multiplications, as we can factorise x[0]\*b[0] + x[1]\*b[1] + ... x[BUFSIZE-2]\*b[BUFSIZE-2] + x[BUFSIZE-1]\*b[BUFZSIZE-1] into b[0]\*(x[0]+x[BUFSIZE-1]) + b[1]\*(x[1]+x[BUFSIZE-2]) + ...

This is achieved by initialising an index, j, to start at the newest data value, and another index, k, to start at the oldest data value. Every iteration, j will increment and k will decrement, and the convolution is carried out as described by the factorisation above. Again there is the issue of under/overflow, which is mitigated by two if statements that work similarly to the one in circ\_FIR1

If the number of coefficients is odd, then j and k will not reach the middle element, since the condition of the for loop specifies to break before it is reached. Due to this, we manually add the last value to the result. It would be equally correct to allow the middle element to be added twice and then subtract it once afterwards, but this would require an extra unnecessary calculation, so the former method was chosen.

circ\_FIR5 : //Optimised verison of 1 (normal circular)

This is an improved version of circ\_FIR1 – instead of array indexes, pointers are used. To understand why this is more efficient, we must realise that an array index is composed of two parts: a pointer to the start of the array (the name of the array) and an offset from the start of the array (the number in square brackets). When the compiler evaluates an array index, it has to compute the position by adding the offset to the pointer, which costs cycles. What we do, instead, is to post-increment the pointer every time it is used, ensuring that it is already at the correct position when it is needed, which cuts the time that was being used for the index evaluation. Additionally, we make use of the register keyword: memory accesses take a lot of time, and the register keyword forces the variable to be kept in a register so that it is always easily accessible by the processor. This is especially useful when applied to the variable storing the accumulated result of the convolution, as it is accessed every iteration of the loop