Part 1-Console Input

The first section of the program takes input from the console and stores it as a number. An important function here is the *getchar* function, which takes input from the keyboard and stores the corresponding ASCII character in R0. R0 can only store one character at a time, so the program must have a way of combining the individual input characters into the whole number they represent.

Read

The *read* function is looped through in order to build up the correct number into R4.

Getkey takes a character from the keyboard and stores it in R0. The program assumes that only numbers have been input, not letters etc.

Lines 13,14 set up a while loop which runs until the enter key is input (ASCII value 0x0D).

By line 16 we have an ASCII value in R0. Since the ASCII value for '0' is 48, R0 minus 48 gives us the number it represents.

Since we know that the user hasn't pressed RETURN, they must have entered the next rightmost digit of the number.

Lines 18,19 multiply the current value in R4 by 10, then add the new digit/character, storing the result in R4.

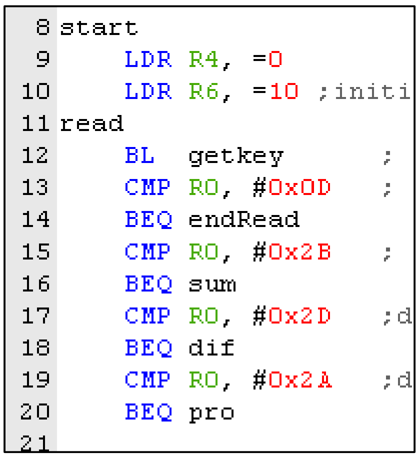
This neatly allows for conversion of an input number to its actual value.

Part 2-Expression Evaluation

The second section of the program is designed to evaluate an input expression of two operands and one operation of addition, subtraction or multiplication.

This builds upon the Console input program.

The program at first runs just like Console input, taking in numerical characters an converting them to a number stored in R4. However this time, if/when the user inputs an operation character (+,-,*), the program branches to a section which deals with this.



```
 8 start
 9       LDR R4, =0
10       LDR R6, =10 ;initi
11 read
12       BL  getkey      ;
13       CMP R0, #0x0D   ;
14       BEQ endRead
15       CMP R0, #0x2B   ;
16       BEQ sum
17       CMP R0, #0x2D   ;d
18       BEQ dif
19       CMP R0, #0x2A   ;d
20       BEQ pro
21
```

Lines 14-20 are what allow this: during each loop of read, as well as checking for RETURN, the program checks if the characters were operations by using their ASCII codes.

If true, we branch to one of the functions *sum*, *dif*, or *pro*. These functions do two things:

1. Since we know that the first whole number has now been input, it can be stored in another register (line 33).

2. These functions store a value in R10 simply marking that the operation is +,- or *. This can be used later when we are actually evaluating the expression.

The program then branches back to start, and the character reading begins all over again, continuing to store the second value in R4 until RETURN is entered.

Once RETURN is entered, the expression can begin to be evaluated. We branch to *endRead*, and print an equals sign. How the expression is evaluated is determined by the value stored in R10 by *sum*, *dif* or *pro.* 0 represents addition, 1 subtraction, 2 multiplication.

```
If R10 == 0
        R5 = R7+R4
```

or

```
If operation == addition:
        Result = First operand + second operand
```

And thus we have evaluated the expression.

Part 3-Console Display

This final section completes the program by displaying the calculated value in R5. The key function here is *sendchar*, which takes an ASCII value as an input and outputs the corresponding character on the console.

This poses a problem: the number produced by the calculator could be many characters in length. To solve the problem, the program manipulates the value in R5 in such a way as to extract the leftmost digit from the number, input the corresponding ASCII value into *sendchar*, then repeat for the remaining number until the remaining number is 0. How this is implemented is shown below.

NB: the part 3 code begins at line 56 of ConsoleDisplay.

Division

The method used to break the number into its digits requires division.

The program divides numbers using this pattern:

```
//for a/b
remainder = a ;
while ( remainder >= b)
{
quotient = quotient + 1;
remainder = remainder − b ;
}
```

This method is required to be used twice, from lines 62-77 in the *findlength* function,

and in lines 96-103 within the "*n modulo 10^count*" function.

```
56      LDR R9,=0
57      CMP R5,#10
58      BMI endLength
59
60
61      MOV R8,R5
62      LDR R1,=10
63 findLength
64      MOV R2,R8
65      LDR R3,=0
66 subtract
67      CMP R2,R1
68      BMI endSubtract
69      ADD R3,R3,#1
70      SUB R2,R2,R1
71      B   subtract
72 endSubtract
73      ADD R9,R9,#1
74      CMP R3,#10
75      BMI endLength
76      MOV R8,R3
77      B   findLength
78 endLength
```

Part 3 begins on line 56, where a new variable is created.

By dividing n by 10 until the quotient is 0, and incrementing count, count can represent n's length minus one.

By line 77 the division(s) by 10 are over and count is stored in R9.

Since we now essentially know the length of the number, all that remains is to divide the number by decreasing powers of 10, beginning with 10^count, printing the quotient each time, until the remainder is 0, thus printing out the number.

This works (and handily removes leading zeroes) because the magnitude of a number length *l* is 10^***l***. So that number modulo 10^(***l***-1) must give the first digit.

This is what the rest of the code does, except it stores the remainder of the modulus/division so that the process can be repeated on this remainder, until remainder = 0.

Dividing by decreasing powers of 10

This is performed within the *nModTenLength* function. As the program executes, count is (during any iteration of nModTen) some value *c*, which represents the magnitude of the current number (ie: the remainder of the last iteration).

The following pseudo-code shows how the program calculates the required power of 10 on lines 87-99:

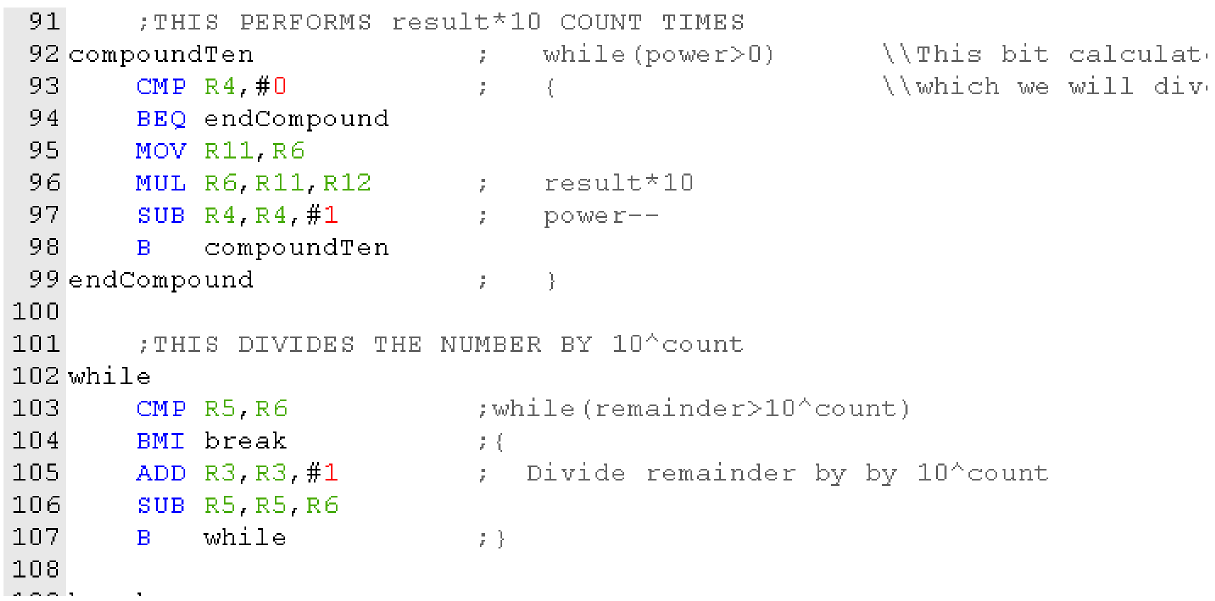
```
power=count
result=1
while(power>0){
result*10
power--
}
```

Once the required power of 10 is calculated this can be subtracted repeatedly from the input number (ie: number%10^count)(see Division).

The quotient of this operation will be the leftmost digit of the input number. The remainder will be the digits to the right of this, and is stored to be reused upon iteration.

See lines 102-109

R6 stores the power of 10, R5 stores the input value, R3 stores the quotient.



```
 91     ;THIS PERFORMS result*10 COUNT TIMES
 92 compoundTen                 ;    while(power>0)       \\This bit calculat
 93     CMP R4,#0               ;    {                    \\which we will div
 94     BEQ endCompound
 95     MOV R11,R6
 96     MUL R6,R11,R12          ;    result*10
 97     SUB R4,R4,#1            ;    power--
 98     B   compoundTen
 99 endCompound                 ;    }
100
101     ;THIS DIVIDES THE NUMBER BY 10^count
102 while
103     CMP R5,R6               ;while(remainder>10^count)
104     BMI break               ;{
105     ADD R3,R3,#1            ;   Divide remainder by by 10^count
106     SUB R5,R5,R6
107     B   while               ;}
108
```

Printing Quotient

Finally, we reach the section of the function nModTenLength we've all been waiting for: printing the digits!

As shown above, R3 holds the quotient of the division, which will be a single character. This is copied to R0, so that it can be used by the *sendchar* function.

It is converted to its ASCII value by adding 48, and sent off to the console to live happily ever after. We then decrement count (because the next number to be input into the *nModTenLength* function is one digit shorter: the remainder), and branch back to the start of the printing function, unless count has reached -1, (lines 117,118) in which case we have printed all the characters and the loop terminates, ending the program.

Methodology

Below is a table showing my inputs to the program, and what the output was.

	Operand 1	Operation	Operand 2	result
1	0	+	0	0
2	1	+	0	1
3	1	+	1	2
4	-1	+	1	2
5	9	+	1	10
6	0	-	0	0
7	1	-	0	1
8	0	-	1	0
9	0	*	0	0
10	1	*	0	0
11	1	*	1	1
12	1	*	2	2
13	10	*	20	200
14	-1	*	1	1
15	-1	*	-1	0
16	4294967296	+	0	0

1. This is correct and shows that the program works for this trivial case.
2. This is correct and shows that the program can add a positive number and 0
3. This is correct and shows that the program can add two positive numbers
4. This shows how the program was not designed to handle negative operands.
5. This is correct and shows that the program can evaluate expressions which require carrying (ie: increase in digits).

   Both 6 and 7 work as expected, showing that simple subtraction works. However:

8. This demonstrates the limitations of the program: it was not designed to handle negative results.

9. This shows that nothing goes wrong at the lower limit of the operands when multiplying.

10 shows that the program correctly produces 0 when multiplying a positive number by 0.

11 and 12 show that multiplication works for small positive values

13 shows that operands greater than 9 can be multiplied to correctly give many-digit numbers.

14. This shows once more how the system cannot handle negative numbers.

16. 4294967296+0=0. This demonstrates that the size of the registers is limited.

Explanation of incorrect results.

The incorrect results from the above testing arose from using negative numbers (apart from the register overload). The program was not designed to handle this.

Test 4 seemingly ignores the negative sign on the first operand, and just adds 1 and 1.

This is because the program overwrites R10 every time an operation is input, and thus the operation performed becomes the latest input.