Simple Encryption

All communications in public space should really be encrypted.

Various forms of encryption exist and at their simplest they used a “shared secret” - a key string that is used to encrypt and decrypt a message using mathematical progression. Such encryption is termed “symmetrical” as the same key is used to both encrypt and decrypt. It is crucial that the key remain secret for obvious reasons.

Stronger, asymmetrical, encryption uses two separate keys - a public key used to encrypt the data and a private key for decrypting. The public key cannot be used to decrypt the message and so it does not matter that it becomes openly known or shared (in fact this is encouraged or even necessary). The private key must remain secret in the same sense as that for symmetrical encryption.

Encryption is further enhanced by adding a “SALT” to the key such that it is different each time but the SALT must be communicated somehow so that the decrypting device is working with a valid key. Further, the SALT should not be easily guessable (such as the time) or even used more than once. This remains a challenge on small devices with custom or no Operating System. The routines below do not use a SALT but you could easily adapt them to do so.

The RC4 Encryption and Decryption Functions found elsewhere in this library provide a standard form of symmetrical encryption and although RC4 is deprecated it still provides a good level of secrecy with a sufficiently complex key of at least 32 characters (256 bits).

Encryption suites such as RC4 are mathematically intensive and smaller micro-controllers, especially running at lower clock speeds, can struggle to maintain timings in tight spaces purely because it can take a good deal of time to generate the encrypted string or regenerate the original. Doubly so where an encrypted reply is required.

If only simple obfuscation is required, you might consider Base64, but note that is very easily “un-done”.

The below functions use a simple rotating XOR of a shared secret key and have the advantage of not being weakly-obvious nor computationally intensive, resulting in fast execution and reasonable custom encryption - not to the level of heavyweight, standards but certainly enough to prevent clear-text snooping-on-traffic-with-easy-reveal.

A major flaw with the below is that is that the encryption shows traits for similar messages that can provide clues to a reasonably competent hacker (rare and not likely listening to the chatter from your air-con controller). Consider the following:

```
THE QUICK BROWN FOX
6A7C641129613870311469656A633519797A04
j|d)a8p1iejc5yz

the quick brown fox
4A5C441109411850111449454A431519595A24
J\D     APIEJCYZ$
```

Even to the untrained eye, it is obvious the two outputs bear a similarity, especially that case seems to track the opposite of the input. This is not entirely why passwords on good computer systems insist of a mixture of both upper and lower case characters, but it serves to illustrate the point. If simple obfuscation is required to stop the casual, unskilled snooper, these two functions will probably be good enough but DO NOT use them where good security is required.

Examples

z$=Encrypt$("mary had a little lamb") z$=Decrypt$(mymessage$)

The Code

```
    'optimized version 12NOV2021
    'change the key to whatever you want but the longer the better
    Const
key$=">4!"+chr$(202)+"1x+chr$(188)+4+chr$(240)+3z4+7%4{9?5\3+chr$(222)+^+chr
$(221)+5$9=6@1~6,+chr$(193)+|1)7:8<3*8S9I9l7Z1eT0r1"

    Function Encrypt$(a$)
        Local Integer n,p,t
        Local b$
        p=1:t=Len(key$)
        For n=1 To Len(a$)
            b$=b$+Hex$(Peek(Var a$,n) Xor Peek(Var key$,p),2)
            p=p+1:If p>t Then p=1
        Next
        Encrypt$=b$
    End Function

    Function Decrypt$(a$)
        Local Integer n,p,t
        Local b$
        p=1:t=Len(key$)
        For n=1 To Len(a$) Step 2
            b$=b$+Chr$(Val("&h"+Mid$(a$,n,2)) Xor Peek(Var key$,p))
            p=p+1:If p>t Then p=1
        Next
        Decrypt$=b$
    End Function
```