

Beacon Timestamp

Andy Talbot G4JNT November 2006

A proposal allowing automatic QSL information to be appended to beacon transmissions.

With the new generation of beacons coming on stream that make use of GPS or similar timing to control their transmissions, it is now possible to add a timestamp to the transmission that appears random to the casual listener, but can allow independent verification of the beacon's reception at any given point in time, allowing an automatic QSL to be generated. The timestamp appears as a group of letters appended to the beacon ident (usually be sent in CW), and is generated from the date and time information delivered from the GPS receiver, changing each minute. If a listener logs the received timestamp with the date and time of its reception the beacon keeper, or any other authorised person who knows how the timestamp was generated, can check the authenticity of the report by calculating the timestamp that should have been sent for the actual date/time of the logged report and comparing with that actually heard.

The timestamp consists of three letters, called a triad here, formed of a consonant-vowel-consonant combination. This combination leads to a 'pseudo-word' that in most cases will be pronounceable and makes the logging process a little bit easier. To prevent the international distress message SOS from ever appearing, the letter S is excluded from the set of letters used. Also, Q has been left out to prevent any confusion with Q codes, and because words with Q's in them are not very easy to pronounce! Y has been assumed to be a vowel.

For this scheme to work properly, it must be impossible for any listener to be able to predict, in advance, what code will be sent for any future particular date/time combination, even if many (or all) past combinations are known. Also, the authorised checkers must be able to predict in advance all codes that will be sent. Where several beacons employ this scheme, each must produce a different sequence of triads to prevent the transmission from one beacon being used to falsely authenticate that from any other. Authorised checkers for any one particular beacon transmission, who will know their own codes in advance, must not be able to predict those for any other transmission from any other beacons outside their authority.

The consonant-vowel-consonant structure (without Q and S) leads to a maximum of $18 * 6 * 18 = 1944$ different three letter combinations, so triads are likely to be repeated from time to time – especially as $60 * 24 = 1440$ different codes will be sent each day. However, repeatability of any particular triad will appear to be completely random and if there are any repeats of pairs (or more) of triads, this should only ever occur only as frequently as the laws of probability allow. The simple nature of the transmitted code does not detract from the overall security of this time stamping scheme

Timestamp Generation

A process is needed to generate a different sequence of apparently random timestamps for each transmitter, but to keep a common algorithm / software package for ease of implementation and can be replicated by the authorised person. The solution is to use a key-based approach where a fixed binary key is chosen for each transmitter. The key is then mixed with the time/date information every minute, then shuffled and compressed to generate the three-letter timestamp code. The key is different for each transmitter, is known only by the authorised person and is stored in non-volatile memory as part of the beacon keyer software.

It must be possible to generate the timestamp in a simple microcontroller such as a PIC, from the GPS receiver output supplied via the NMEA data or similar, so the algorithm must work with single bytes at a time. The timestamp algorithm must be widely published and the security of the scheme must not depend in any way on the algorithm itself, but only on the key. (See Ref 1)

The key length has to be long enough that anyone with programming skills cannot guess, or be able to derive it from past information.

Algorithm Description

Generate a 64 bit key which is unique for each beacon and/or site; there should be no weak keys and, for example 0x00000000 should give results apparently as random and any other value.

Split this into eight key bytes $K(0) - K(7)$

Each minute, read the time and date and compress into four (Plaintext) bytes $P(0) - P(3)$ as follows :

P(0) = Minute	(000mmmmm)
P(1) = Hour	(0000HHHH)
P(2) = Day + 32 * (Month MOD 8)	(MMMDDDDD)
P(3) = Month \ 8 + 2 * (Year MOD 128)	(YYYYYYM)
<i>(alternatively, using C-like terminology)</i>	
P(2) = ((Month << 5) AND 0xE0) + Day	
P(3) = ((Year << 1) AND 0xFE + ((Month >> 3) AND 1)	
Use the full 4 digit Year, eg 2006.	

Initialise two working registers with $X0 = 0x2B$, $X1 = 0x89$

Generate the encrypted time stamp by passing through the algorithm shown in Figure 1, eight times. $X0$ and $X1$ inputs for each pass are the results $X0'$ and $X1'$ from the previous one. Eight passes through the algorithm are enough to spread out a single data bit over all 16 bits of the final word. The indices for $K(i)$ and $P(i)$ are repeatedly cycled from 0 to 3 and 0 - 7, respectively, for each pass. Note that the instructions labelled << are byte rotations which preserve all bits present, they are not shifts that would lose any bits shifted out.

From $X0' / X1'$ generated after the final pass, form three pointers into two lookup tables:

C1	=	$X0' \text{ AND } 0x1F$	(0 - 31)
C2	=	$(X0' \text{ AND } 0xE0) \setminus 32$	(0 - 7)
C3	=	$(X1' \text{ AND } 0x7C) \setminus 4$	(0 - 31)

Use a lookup table containing 32 consonant and 8 vowel options. Letters are repeated to extend the tables to binary lengths.

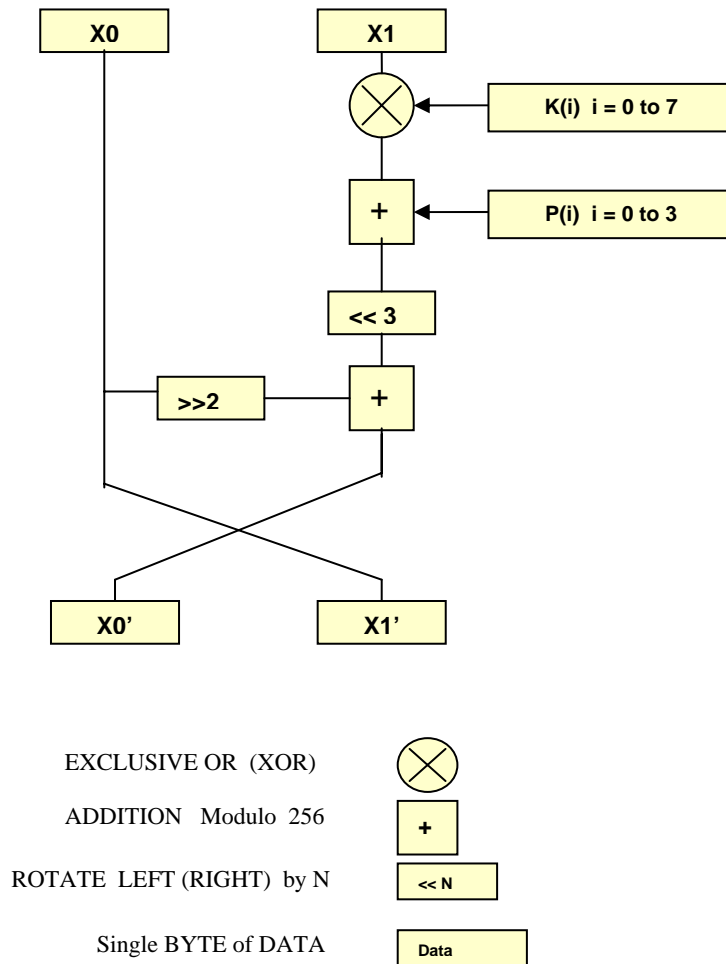
<i>CONSONANT_TABLE</i>	=	"BCDFGHJKLMNPRTVWXZBCDFGHJKLMNPRTW"
<i>VOWEL_TABLE</i>	=	"AEIOUYAE"

The triad is then formed from:

CONSONANT_TABLE(C1) & VOWEL_TABLE(C2) & CONSONANT_TABLE(C3)

This algorithm can be programmed into the 16Fxxx family of PICs using less than 100 words of assembler for the hashing code, and executes in around 340 clock cycles. Annex A shows the PIC 16Fxx assembly code.

Figure 1 – One iteration of the Timestamp Hash Algorithm.



Security of the Algorithm

The process of mixing the plaintext data (the date/time) with the key data to generate an unknown hash (the timestamp) has been borrowed very loosely from standard cryptographic techniques, particularly those used in DES and TEA (Ref 1). Originally a 16 bit key was chosen for simplicity, until James Miller G3RUH pointed out that any hacker could easily try all 65536 combinations of possible key against just a few known date/time/triad combinations until a match is found, revealing the key which would allow all future triads to be predicted. This process would take a few milli-seconds at most on a modern PC.

A revised algorithm with a 32 bit key was written, and sent to James for evaluation. By writing customised cracker software for running on a 233MHz ARM based computer, he was able to derive the 32 bit key for any four date/time/triad combinations in approximately ten minutes by

cycling through all possible codes. A routine written in machine code for a 2.4GHz PC ought to be able to manage it in less than a minute. Hence the adoption of a 64 bit key here which extends the time taken to crack to a few thousand years, even if PCs do get seriously faster in the near future!

I have no idea at all if the algorithm itself has any weaknesses in it that reduce the effective code length – perhaps someone out there with a background in cryptanalysis can work out any potential faults!

Encryption in Amateur Radio

The amateur licence, enshrined in BR68, specifically states that messages must not be in secret code or cipher, ie. encryption is not permitted for any amateur transmissions. It is important to note here that the technique described is not encryption, in the sense that the contents of the transmission are not concealed – in fact the contents of the timestamp are known exactly; they are a compressed version of the current date and time. The algorithm is open to all to see and it is only the process of getting there (the key) that remains known only to the authorised keepers. The content of any transmission is known at any time.

In fact, the process described is not encryption at all, it is a form of hash generation (See Ref 1 for more details of Hashing). True encryption would require that it be possible to reverse the process and regenerate the plain text (the full date and time) from the three letter timestamp. Theoretically it may be possible, having a run of several timestamps and knowing the key, to be able to recover the time/date when they were formed. I haven't the faintest idea how to start doing that, other, perhaps, than by a brute force attack trying multiple keys until something works.

Support Software

Two PC utilities support the use of this timestamp process. Both run in a Command Prompt.

TIMECODE.EXE takes the 64 bit key as input, with the date , and generates a file named TIMECODE.TXT containing all timestamps for each minute of that particular day. The key may be entered either as Hex characters, or via the password generating process used in the next utility. The programme also allows individual timestamps to be generated for any particular time subsequently entered from the keyboard.

MAKEKEY.EXE generates a 64 bit key from a more easily remembered password. This also employs a hashing technique but the details are beyond the scope of this note. It can be found by looking at the source code (16 bit Power Basic) supplied for both utilities.

Ref 1 Applied Cryptography: Protocols, Algorithms, and Source Code in C.
Bruce Schneier. Second edition, 1996. Published by John Wiley and Sons

Annex A

PIC 16Fxxx Assembler Routines for generating the Timestamp Hash

```
;-----
CompressTime      ; Takes Year, Month, Day, Hours, Minutes bytes.  nb. Year is the low
movf  Minutes, w  ; byte only of the full 4 digit year expressed as a 16 bit integer.
movwf P0          ; Generates compressed date/time in P0 - P3
movf  Hours, w
movwf P1

rlf  Month, w    ;
movwf P2         ; Month * 2
rlf  P2          ;
rlf  P2          ;
rlf  P2         ; P2 = Month * 16
rlf  P2, w       ; W = Month * 32 but loses MSB
andlw 0xE0       ; W = (Month MOD 8) * 32
addwf Day, w
movwf P2

bcf  STATUS, C
rlf  Year, w     ; W = 2*(Year MOD 128)  Loses MS Bit of the low byte.
movwf P3
btfsc Month, 3 ; Month \ 8
incf  P3        ; P3 = Year mod 128 + Month \ 8
return

;-----
Hash              ; Takes 4 bytes of Plaintext P3/2/1/0, and 8 byte Key in EEPROM at address KeyData
                  ; P must be stored in increasing memory P0,P1,P2 etc as it is addressed indirectly
                  ; Generates pronounceable triad in C1/2/3.  Uses X0/1

movlw HIGH(Vowel) ; Segment where consonant & vowel tables are stored
movwf PCLATH

movlw 0x2B        ; Seed values
movwf X0
movlw 0x89
movwf X1
clrf Counter

HashLoop
movf Counter, w
andlw 0x07       ; 64 bit key,
sublw 7         ; W = 7 - W. Converts count from 0-7 to 7-0,
                ; backwards for Big-Endian readability of data in EE
addlw LOW(KeyData) ; Point to start of Key data in EE
call  GetEE     ; Return with data in W
xorwf X1

movlw P0         ; Address of beginning of Input data
movwf FSR
movf Counter, w
andlw 0x03
addwf FSR        ; Point to P0 - P3 as required
movf INDF, w
addwf X1

rlf  X1, w      ; 3 bit rotate, has to use pairs of RLF's coz of the carry
rlf  X1, w      ; - very annoying!
rlf  X1, w
rlf  X1, w
rlf  X1, w
rlf  X1, w

movf X0, w
movwf Temp

rrf  Temp, w    ; 2 bit rotate
rrf  Temp
rrf  Temp, w
rrf  Temp, w

addwf X1, w
movwf Temp

movf X0, w
movwf X1
movf Temp, w
movwf X0

incf Counter
movlw 8
subwf Counter, w
btfss STATUS, Z
goto HashLoop

; ----- Now generate the pronounceable triad -----
```

```

movf    X0, w
call    ConsonantTable ; Truncation to 5 bits is done later.
movwf   C1

movf    X0, w
movwf   Temp
rrf     Temp
rrf     Temp
rrf     Temp
rrf     Temp
rrf     Temp, w
call    VowelTable
movwf   C2

movf    X1, w
movwf   Temp
rrf     Temp
rrf     Temp, w
call    ConsonantTable
movwf   C3

return          ; ASCII values stored in C1 C2 C3
;-----
; Ensure both are in the same segment
VowelTable
andlw   0x07
addwf   PCL
dt      "AEIOUYAE"

ConsonantTable
andlw   0x1F
addwf   PCL
dt      "BCDFGHJKLMNPRTVWXZBCDFGHJKLMNPRTW"
;-----

```