FSQ and Error Correction

Describing how the impossible can be achieved through the application of a series of interesting technological steps.

Murray Greenman ZL1BPU
August 2016 V0.4

Introduction

FSQ (Fast Simple QSO) is an extraordinary and relatively new method of transmitting text-based intelligence by radio. Unlike almost all previous modes, which are asynchronous (RTTY, Morse) or synchronous (PSK31, MFSK16, AX25), the FSQ modem is *non-synchronous*. Its reception is neither timed as in RTTY from a particular event at the start of each character (symbols are sampled at fixed times from this point), nor sampled according to a recovered clock applied to all symbols (PSK31). No, FSQ does neither - reception is determined by discovering when a new symbol has been perceived, then waiting a certain time, typically three further samples, to ensure that the new symbol is firmly established, before sampling it.

This non-synchronous modem gives FSQ extraordinary performance in the presence of serious reception timing errors, which was the primary motivation for its development, namely to provide reliable digital communication under Near Vertical Incidence Signal conditions (NVIS), prevalent on HF bands between 2 MHz and 10 MHz. Under these conditions, signal arrival times at any moment can vary by hundreds of milliseconds.

Another property of FSQ (use of a 33 tone MFSK modem) allows each lower-case letter or common punctuation character to be sent using just one symbol.[1] Thus the typing speed is remarkable, despite using slow symbol rates (the latter giving the mode enhanced sensitivity and robustness, as well as the tolerance of timing errors just mentioned).

While FSQ has remarkable performance that generally does not justify the use of error correction, there are occasions, especially in Emergency Communications and Public Service applications, where absolute accuracy is important – passing addresses, phone numbers, personal details etc. Unfortunately when errors occur in FSQ, they tend to be of a type which is not easily corrected using conventional convolutional coding or block code Forward Error Correction: rather than the usual letter or bit *substitutions*, FSQ is uniquely prone to symbol *insertions* and *deletions*. Until the work described here, the only error protection possible in FSQ was to use error detection techniques such as check sums and cyclic redundancy checks (the latter are used to protect call signs in headers), and no automatic correction was possible.

There are however occasions where even the most trivial errors cannot be tolerated. FSQ is used by the FSQCall selective calling and messaging system, designed specifically with Emergency Communications as a major application. If a message were to be sent containing personal details, phone numbers, detailed instructions, or even medical prescriptions, it would be useful to know that the message received is *guaranteed* to be correct. This is where this development started: to provide a way to send files securely - and to know absolutely that the received file is 100% correct.

[1] The *symbol* is the smallest unique item in a digital radio transmission, typically a bit (PSK or RTTY) or a group of bits (MFSK16, DominoEX). In FSQ the symbol, one of 32, can represent a complete lower case letter.

How Errors Arise

In digital mode radio reception (in the case of MFSK and PSK reception at least), most errors consist of replacement of the value of a symbol or series of symbols with incorrect values. The errors are induced by noise when the signal is weak, by burst noises (ignition noise, lightning, splatter), and by momentary loss of reception (lightning induced AGC suppression, carrier interference, fading). In most modes the symbols are simply replaced by erroneous ones, but the same number of symbols remain, since the signal is sampled synchronously. This is important, as error correction algorithms (especially the block type) cannot tolerate insertion of extra data or deletion of data.[2]

FSQ uses letter-based symbols, one for each lower case letter and most common punctuation, and two sequential symbols for all other characters. The second symbol (or lack of it) defines which of four code tables the first symbol refers to. The mode is subject to errors of substitution, as with any other mode, but has two unique problems: *insertions* into and *deletions* from the data stream, and although not common, these need to be considered.

Insertions arise when one of the two-symbol characters is misinterpreted as two single-symbol characters. Deletions arise by the reverse mechanism - two single-symbol characters are interpreted as a pair. Thus far no Forward Error Correction technique (FEC) has been discovered which could deal with such abuse of the data stream. Block error correction techniques such as Reed-Solomon, really the only practical type for a character-based modem, perform very badly with these types of errors.[3]

The Plan

Not wanting to deviate from the FSQ modem, the strong NVIS performance and the ability to send one character per symbol, something had to give in order to eliminate the insertions and deletions. It was decided that the receiver absolutely had to operate synchronously during file reception. This isn't a design back down, but a way to enhance the capability of the mode. Under normal reception, even file reception, normal non-synchronous FSQ operation remains in operation, and no change is made to transmission (which, having fixed duration symbols is essentially synchronous anyway).[4]

For FEC correction of files, the receiver switches on an extra capability, namely that of recording the MFSK detector output to file at every sample, 48 times per second. The FSQ MFSK detector uses an overlapping Fast Fourier Transform, with a solution and decision made at this 48 Hz rate. The output is a bin number (in the approximate range 400 - 600), the FFT bin in which, for the moment, the most signal power is concentrated. When file reception is triggered, this bin data is recorded at a fixed rate, and once reception has stopped, the data is saved to file, to be analysed off-line.[5]

The 'raw' file, recorded at 48 samples per second, contains 12 samples for every symbol at 4 baud (16 samples at 3 baud etc), but contains no information about where the symbol boundaries are located, only the bin numbers at each sample.

[2] This is in contrast to 'puncturing', which is a powerful technique used to replace known faulty symbols with benign ones, while the same number of symbols remains intact.
[3] Previous character-based modes such as DominoEX reverted to a synchronous binary-coded data stream for FEC operation (the mode THOR), abandoning the character-based alphabet used without FEC.
[4] However, the clever ability of FSQ to operate over a range of symbol rates without prior knowledge is lost for the process of synchronous sampling. FSQ error correction always operates at 4 baud.
[5] In other words quite complex analysis of the file data is made possible, since time to provide a solution is no longer confined by the need to maintain the reception process.

The beauty of this concept of saving the raw data is that a completely separate off-line 'helper' program, which need not even be running when the file was received, can handle the decoding of the FEC message file. The helper can also be written in a different language, say one that has strong string handling and array capability. Further, as research and development continues, the performance of the helper program can potentially be enhanced or adapted to new techniques without necessitating changes to the highly complex real-time FSQCall program, or require new training of operators.[6]

Synchronous Reception

First Step - Discovering Sync

FSQ operates using 33 tones, switching between selected tones at a constant rate using a phase-coherent technique.[7,8] As mentioned above, while it is known how many samples the raw file contains for each transmitted symbol, the location of the symbol boundaries or transitions is not known. Further, these transitions are typically blurred[9] by multi-path propagation. Real-time FSQ reception works from a transition and waits for the symbol to be established for several consecutive samples before making a decision, but in a synchronous off-line process it is possible to do even better than this, since we have plenty of time.

An analysis of the received raw sample file is first undertaken to discover where most of the symbol transitions occur. While easy to do visually, by simply studying the numbers in the file, a computational solution requires something extra: in this case a *cross-correlation* technique is used. First we find all the transitions, by subtracting each sample from the previous one, and squaring the result. The answer will be mostly 0, but where a symbol changes, there will be a frequency difference at the transition, maybe as small as two or three bins, maybe much larger. Here's an idealized example. First, a list of received tone samples:

505, 505, 505, 511, 511, 511, 499, 499, 499

where the differences are:

... 0, 0, 6, 0, 0, -12, 0, 0, ...

and the differences squared are:

... 0, 0, 36, 0, 0, 144, 0, 0, ...

so it becomes very obvious where the symbol transitions are. In practice the transitions are further (12 samples) apart than in the examples given, and there are also small transitions found during the symbols as a result of noise, drift and Doppler effects. Thus when determining the transition peaks, a method needs to be employed that either ignored these minor effects, or renders their randomness inconsequential.

[6] For example, a means to 'learn' the symbol rate might be added, or a way to realign the sampling part way through the file might be added, in order to handle extreme conditions or errors in sending speed.

[7] The switching point between tones (symbols) is called a '*transition*'.

[8] *Phase coherent* means that when the tone frequency is changed, it does so without any abrupt change of phase. This is an inherent property of Direct Digital Synthesis.

[9] Both moved in time from symbol to symbol, *and* delivered with multiple different transition points via different ionospheric paths.

Now in order to know where these peaks (possible symbol transitions) are in relation to the whole file, we need to consider that:

(a) Individual peaks can move around in time (as much as several samples) due to propagation effects.
(b) There can be spurious peaks caused by noise and drift.
(c) Some peaks may in fact be missed.
(d) The peaks may occur regularly, but at points unrelated to where the file starts.

It is therefore best to analyse the whole file, in order to collect the information about peaks in the whole file, determine their relationship to the start of the file, and average out the anomalies.

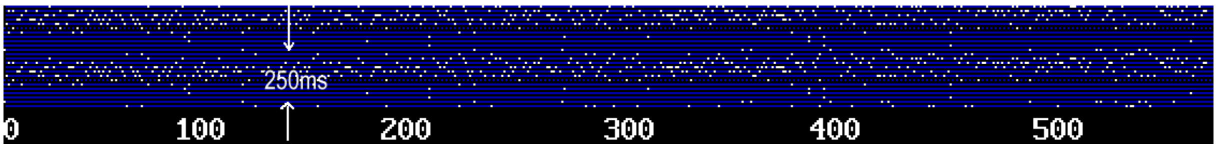


Fig.1. Typical symbol timing

Figure 1 shows an analysis of the received signal timing, over a 300km path on a typical evening on 80m. Each white dot represents when a symbol transition occurs. The vertical axis is symbol timing, 250ms being the period of each symbol at 4 baud. There are 12 samples per symbol. The horizontal axis is also time, one symbol per column, and the marked scale is approximately that of characters transmitted. The display is seamlessly repeated twice vertically, in order to have continuity if the transitions occur at one end or the other of the arbitrarily phased 12 samples.

On a direct path, the graph is simply a dead straight line, with perhaps slight drift due to transmitter/receiver sample timing differences. It is quite easy to see that on the path illustrated in Figure 1, there is easily 100ms 'jitter' in the position of the transitions. FSQ, being non-synchronous, simply discovers a transition, records the FFT bin now containing the biggest signal, then waits a certain time (typically three samples, or 60ms) to see whether the same bin is still recording the biggest signal, and so has a real symbol, or maybe just noise. Thus normal FSQ handles this variability of 100ms or so with no great effort.

But in order to sample synchronously, the receiver needs to locate the best spot to sample the signal at a steady rate, i.e. synchronously, so no sample is lost. Clearly, the best place to do this is as far as possible from where the transitions are occurring, i.e. where the least number of white dots are in Figure 1. This is easy to see by eye, but to do this in a computer requires a special technique.

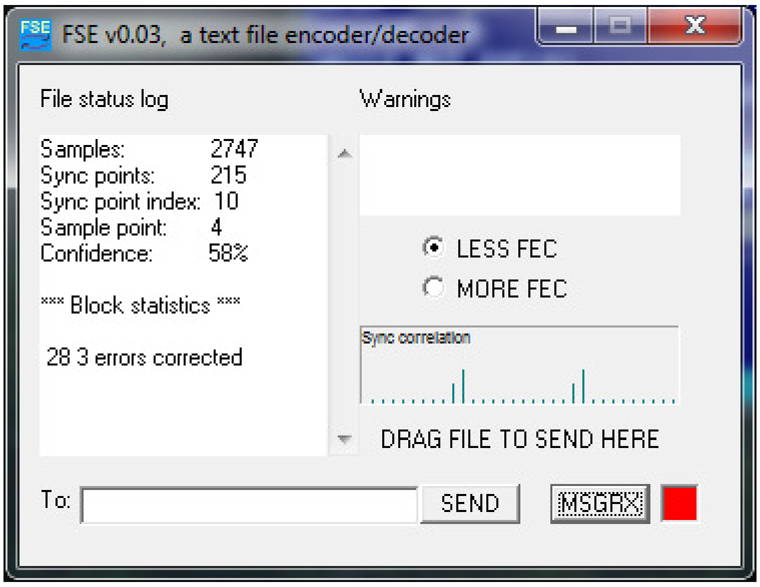
First we make a new file, the difference file, which contains only the differences between samples. Then we analyse this file to work out where most of the transitions occur. Figure 1 is derived from this information.

The technique used to do this is called *cross-correlation*. Since we know the symbol rate before-hand, we know *on average* how often the peaks occur, i.e. how many samples apart. So we can use a simple programming technique to create a buffer of this length (12 bins at 4 baud), and simply record the differences as it examines each sample through the length of the difference file, by adding them sequentially to one bin at a time. For example, the first difference goes in bin 1, the second in bin 2, the third in bin 3, etc. But so also difference 12 is added to bin 1, and difference 13 to bin 2 etc. in other words, the bin *address* is determined *modulo 12*, while the bin content is the sum of all samples in the file with that periodicity and delay.

After we've analysed a 4-baud file containing say 100 symbols (1200 samples), we will end up with a result in the correlator bins something like this:

1, 0, 0, 55, 1103, 42, 0, 0, 0, 0, 0, 0

A transient at the start of the file may cause the '1' in the first bin,[10] but it's very obvious by eye that the peak is in bin 5 (marked in red). This is where *most* of the symbol transitions have occurred. Any of the 12 bins may contain the peak, depending on when the raw file recording started.


FSE v0.03, a text file encoder/decoder

File status log

Samples: 2747
Sync points: 215
Sync point index: 10
Sample point: 4
Confidence: 58%

*** Block statistics ***

28 3 errors corrected

Warnings

[✓ LESS FEC] MORE FEC

Fig.2. FSE screen shot with correlator display

You can see the Sync correlation results illustrated in the above screen shot from FSE. Most of the energy is in bin 10, with some in bin 9 as well. (Count the dots from the left: the graph is repeated twice horizontally).

Cross-correlation is very secure and sensitive. Since correlation is an analogue technique, strong sharp peaks will influence the result more than weak broad peaks, like a 'soft decision' approach, and noise is nicely suppressed, since noise is asynchronous and uncorrelated.

Under conditions of very poor propagation, the transitions tend to move about markedly, and the correlator result can be like this:

1, 2, 22, 55, 636, 408, 77, 0, 0, 0, 0, 0

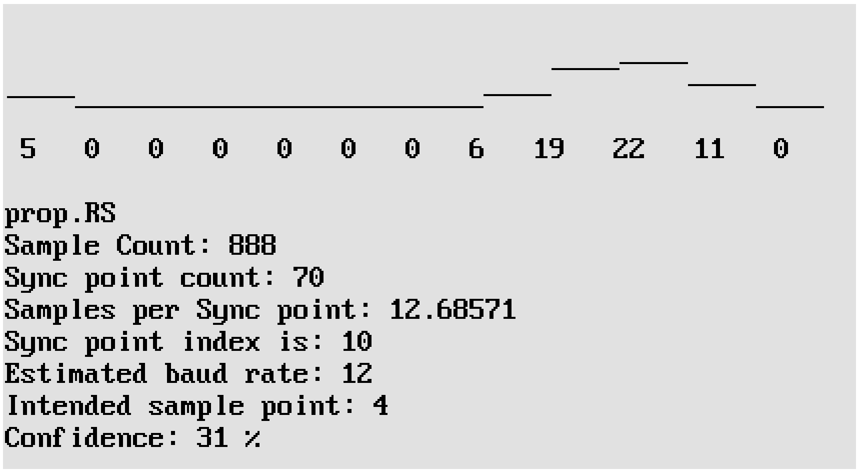
It is clear that considerable blurring has occurred, but the peak is still clearly found in bin 5 (marked in red).

The last step to discovering sync is to determine where to sample the data: clearly (for 12 samples per symbol) this is a position 6 samples *after* or before the transition peak, so the symbols are sampled where least disturbed. This is done by simply identifying the peak bin

[10] The algorithm doesn't know what came before the first symbol so can't determine an accurate difference. The same happens at the end of the file, although there it could add its '1' to any bin.

(in the examples, bin 5) and adding 6, modulo 12, to the index (in the numerical example bin 11). Looking at the correlator results, we can be confident that sampling at bin 11, even under poor conditions, will be as clear of any symbol transitions as possible, even when reception is poorly timed.

Here's a real example:



```
 5   0   0   0   0   0   0   6   19   22   11   0

prop.RS
Sample Count: 888
Sync point count: 70
Samples per Sync point: 12.68571
Sync point index is: 10
Estimated baud rate: 12
Intended sample point: 4
Confidence: 31 %
```

Fig. 3. Screen-shot showing the correlator process

Figure 3 shows the correlator result from some test software used during development. It was used to sample a real data file (from an NVIS transmission) containing about 70 characters. It is quite clear, from the little graph here, that the peak where most transitions occur is quite blurred as a result of variable propagation. The numbers under the little graph show how many transitions were accumulated in each bin. Under perfect conditions, all the energy would be concentrated in one bin.

In Figure 3, the peak is found in bin 10, and the optimum sample point is bin 4, just as the text message in Figure 3 shows.

Mathematically, the peak is discovered by comparing one correlator bin sum with the next, from bin 1 to bin 12, and when the next bin contains a bigger number, that number and the bin number are remembered, giving us the peak value and peak bin respectively.

This peak pin 'Sync point index' (bin 10) is indicated in the metrics in Figure 3. This information is also displayed by the real FSE program (as in Figure 2).

The Confidence figure in Figures 2 and 3 is a measure of the height of the biggest bin compared with the energy in all the bins. Confidence is a simple metric for correlator performance (or rather the quality of reception). This is essentially a measure of the narrowness of the peak. Reception is still perfect at well below 30%.

As previously mentioned, the 'Intended sample point' in Figure 2 or 3 is calculated by adding 6 (modulo 12) to the Sync point index. This is the point at which the symbol data should be sampled, in order to best avoid errors around the transitions. You will note that in Figure 3 the 'Intended sampling point' is 4, and bin 4 and its neighbours clearly contain no symbol transition energy, the perfect spot to sample the symbols.

The larger the file of differences that can be assessed is, the more accurate the Correlation can be, although drift in the sync will ultimately limit this approach, not that it has been found to be a problem with reasonably sized encoded files.

Sampling the Symbols

Now that the transition points have been determined, and a reliable sampling point determined, we go back to the original raw file, and using the sampling point as a starting point index, we sample every 12th FFT sample into a new file.

As you will by now realize, while FSQ uses MFSK, the tones are allocated using an Incremental Frequency-shift Keying technique, or IFK, which is based on adding data for one symbol to that of the next, and subtracting when received. This technique is drift-proof and handles Doppler shift from the ionosphere well.

FSQ uses symbol numbers 0 to 31, and if the subtraction on reception results in a number below zero or above 31, then 32 is added or subtracted to keep it in this range. FSQ also has an up-frequency shift of one step on every symbol, which is taken into account at this point.[11] Thus the new file has a list of actual FSQ symbol numbers used by the originating station to send the file.

Correcting for Drift

Up to this point it has been assumed that the sound card sample rate is exactly 12 times the symbol rate. Unfortunately it isn't quite, and while sampling a short file as just described works well, the sampling point becomes less ideal after 500 characters or so. The tendency is for the sampling point to slowly drift early relative to the actual data.

This has nothing to do with sound card sampling rate error, although this could be a problem if the error was gross. It is simply a quirk related to the different algorithms used to transmit and receive the tones.

There are several complex ways in which this could be corrected. We opted instead to simply skip one sample every 500 symbols (or 3000 samples), which keeps things nicely lined up. Note that it is one *sample* that is skipped, not one symbol, so no data is lost.

FSQ Decoding

This process is superficially no different to non-synchronous decoding, as used in FSQCall, apart from being off-line, and decoding is handled in the same way. In FSQCall, as each symbol number is considered, the next symbol is checked to determine which of the four code tables to use (corresponding to no second symbol, and second symbol values of 29, 30 or 31), and the character determined by looking in the appropriate code table using the first symbol number as the index.

However, in order to suppress insertion and deletion errors, the error coded transmitted file will never contain two-symbol characters, and the receiving FSQ decoder (for files only) deals with only one symbol at a time, replacing any symbol that appears to be an impossible character with a space ‘ ‘. Most importantly, the ***one symbol:one character*** relationship must always be maintained, in both coding and decoding. Having a wrong character is far less a problem than having a missing or extra character. How this is achieved is explained in the next section. Essentially the FSQ decoder in this case has a restricted (lower-case only) character set.

The output plain text message is then written to file. It will only contain lower-case text.

[11] This IFK+1 rotation ensures that there is always a transition between symbols, and markedly reduces inter-symbol interference on multi-path impaired paths.

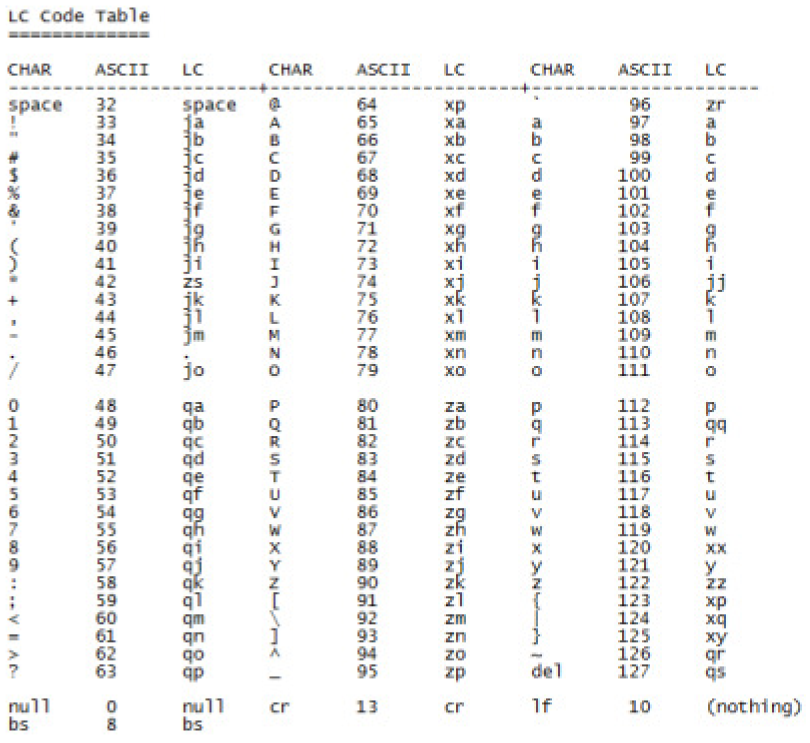
Preparing for Error Correction

The whole aim of this development has been to provide a secure method of reception without insertions and deletions, in order for the FEC decoding algorithm to operate reliably. However, synchronous reception, as just described, still has a flaw, which can still cause insertions and deletions to occur. If an error occurs during reception that changes a first symbol (0 to 28) into a second symbol (29 to 31) or vice versa, we may still have the correct number of symbols, but they do not represent the correct number of characters! Each such error can move the character stream one to the left or right and confuse the character-based error-correcting decoder.

LC Shift[12]

The only possible solution to this is to transmit ONLY lower-case letters, or at least those expressed in a single symbol, as all others are double-symbol. Any errors will therefore be substitution errors, not insertions or deletions. To achieve this with minimal overhead, a shift technique is used, where lower case letters and single-symbol punctuation are sent as-is, except for the four least used letters, 'j', 'q', 'x', and 'z'. These are used as 'shift markers', and so to transmit 'j' you need to send 'jj', or 'z' send 'zz' etc.

The rest of the alphabet is arranged in ASCII column code tables, and sent as lower-case pairs, so for example 'A' might be sent as 'xa', 'B' as 'xb' etc. The same applies to numbers and other punctuation. While this might seem to add considerable overhead, consider this: all these (rest of alphabet) characters are normally sent as two symbols in FSQ anyway, so there is *no difference* in efficiency sending 'xa' compared with 'A'. The only overhead comes in the need to use two symbols for the four least used lower case letters, 'j', 'q', 'x', and 'z'.



LC Code Table

CHAR	ASCII	LC	CHAR	ASCII	LC	CHAR	ASCII	LC
space	32	space	@	64	xp	`	96	zr
!	33	ja	A	65	xa	a	97	a
"	34	jb	B	66	xb	b	98	b
#	35	jc	C	67	xc	c	99	c
$	36	jd	D	68	xd	d	100	d
%	37	je	E	69	xe	e	101	e
&	38	jf	F	70	xf	f	102	f
'	39	jg	G	71	xg	g	103	g
(	40	jh	H	72	xh	h	104	h
)	41	ji	I	73	xi	i	105	i
*	42	zs	J	74	xj	j	106	jj
+	43	jk	K	75	xk	k	107	k
,	44	jl	L	76	xl	l	108	l
-	45	jm	M	77	xm	m	109	m
.	46	.	N	78	xn	n	110	n
/	47	jo	O	79	xo	o	111	o
0	48	qa	P	80	za	p	112	p
1	49	qb	Q	81	zb	q	113	qq
2	50	qc	R	82	zc	r	114	r
3	51	qd	S	83	zd	s	115	s
4	52	qe	T	84	ze	t	116	t
5	53	qf	U	85	zf	u	117	u
6	54	qg	V	86	zg	v	118	v
7	55	qh	W	87	zh	w	119	w
8	56	qi	X	88	zi	x	120	xx
9	57	qj	Y	89	zj	y	121	y
:	58	qk	Z	90	zk	z	122	zz
;	59	ql	[	91	zl	{	123	xp
<	60	qm	\	92	zm	|	124	xq
=	61	qn	]	93	zn	}	125	xy
>	62	qo	^	94	zo	~	126	qr
?	63	qp	_	95	zp	del	127	qs
null	0	null	cr	13	cr	lf	10	(nothing)
bs	8	bs						

Fig. 2. The LC Shift code table

[12] The LC Shift technique was developed for use in the FSQ Encryption program FSPLite (Fast Simple Privacy). Using a reduced alphabet here enabled a vast range of substitution code PIN numbers to be used. Again, single-symbol coding and substitution coding were important for reliable performance in FSQ, which is prone to insertions and deletions.

You can see that the resulting LC Shift code table handles the complete FSQ character set with room to spare. LC Shift coded text is more-or-less human readable. For example 'The quick Brown Fox' might read:

zehe qquick xbrown xfoxx

You might well ask how LC Shift differs from the FSQ alphabet, since many of the characters still require two letters and thus two symbols. It all amounts to *where* the two letters are coded. With LC-shift, the letters are coded and sent through the Reed-Solomon error coding, which is certainly not the case with normal FSQ letter coding, where the alphabet to tone assignment is outside the R-S coding system.

Handling CR and LF

FSQ has only one symbol to mean ‘new line’. The CRLF symbol means both. Text files generally contain both CR and LF at the end of each line, and FSQ sends the CR as CRLF and ignores the LF.

The same has to occur in the LC-shifted text. Techniques had to be added to the LC-shift and un-shift to preserve this relationship. Otherwise, the received file might completely lack new lines, have double new lines, or (perversely) display without new lines in one editor (Notepad) and with them in another (Wordpad).

This requirement is achieved by explicitly converting CR to CRLF, and omitting LF when the LC file is made, and by explicitly converting CRLF into CR and LF when converting back to conventional text. While LF should never be received, if it is, it is ignored. In this way the correct number of received characters is also preserved, allowing the error correction process to operate accurately.

Symbol Table

The symbol table used for file transfer is much reduced, compared to that for regular (real-time) FSQ. Because LC Shift is used, fewer characters are required. The symbol table consists only of the following:

Character	**Symbol**	**Character**	**Symbol**
SPACE	0	o	15
a	1	p	16
b	2	q *	17
c	3	r	18
d	4	s	19
e	5	t	20
f	6	u	21
g	7	v	22
h	8	w	23
i	9	x *	24
j *	10	y	25
k	11	z *	26
l	12	.	27
m	13	CRLF	28
n	14		

* Characters j, q, x, and z are used by the LC-shift technique as shift markers, although they add little overhead to the transmission.

Symbol differences greater than 28 are not used, and if received, are replaced with a single benign character, a space. The symbols are transmitted as differences, in the normal FSQ manner. Despite the reduced transmission character set, using the LC-Shift system, the whole FSQ alphabet can be expressed in the transmitted file.

Reed-Solomon Coding

Without going into details which are necessarily complex, it is sufficient to say that the Reed-Solomon coding technique involves calculation of error syndromes for blocks of text, and sending these ahead of each block of text. The text is still plain text (or nearly so, since it is LC-shift coded), and the FSQCall R-S algorithm has been arranged so that the syndromes are also transmitted using only lower-case letters.

Two strengths of coding are offered, the weaker one with 225 text characters and 30 syndromes per block, the stronger with 195 text characters and 60 syndromes per block. The two modes will correct 15 and 30 character errors per block respectively.

Since the LC-shift characters and the shift markers (see previous section) are also encoded as part of the message, these are also protected by the R-S coding, and thus are immune from insertion and deletion errors.

The LC-shift technique also correctly preserves the CR and LF characters in the original file, so the error correction technique is able to operate accurately on the same number of characters as was transmitted.

The R-S algorithm used is very strong, capable of handling multiple substitution errors with ease, and has very low overhead for an FEC system (60 - 90%). Compare this with convolutional coding methods, which typically have 100% overhead or more.

So the actual on-air message file transmission with LC-shift and R-S coding is efficient and still more-or-less 'human readable'.

Here’s an example of a short message LC-shifted and R-S encoded (stronger mode), ready to transmit:

```
sjhbdgsgtstctrbfwutbeibigctiwshswcfvtiuwhviaasvieautihbdvtuvzlxhelp.txxtzn
xfxiles that xfzdzbxcxaxlxl usesqk

xfzdzbxcxaxlvqaqdqgzpsetup.txxt
xheardlog.txxt
xfiles.bat
etc...
```

The file name is embedded in the file immediately after the syndromes (marked in red). The plain text file (Help.txt) reads:

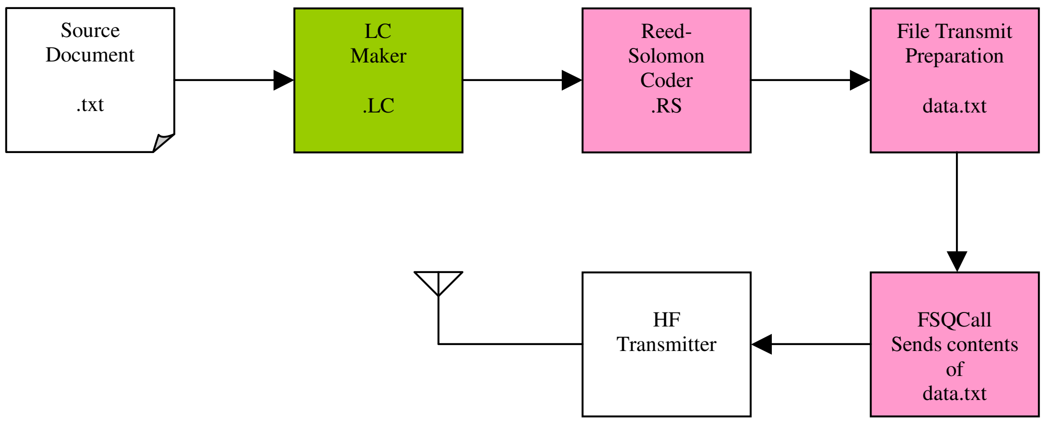
```
[Help.txt]
FIles that FSQCALL uses:

FSQCALv036_setup.txt
Heardlog.txt
Files.bat
etc...
```

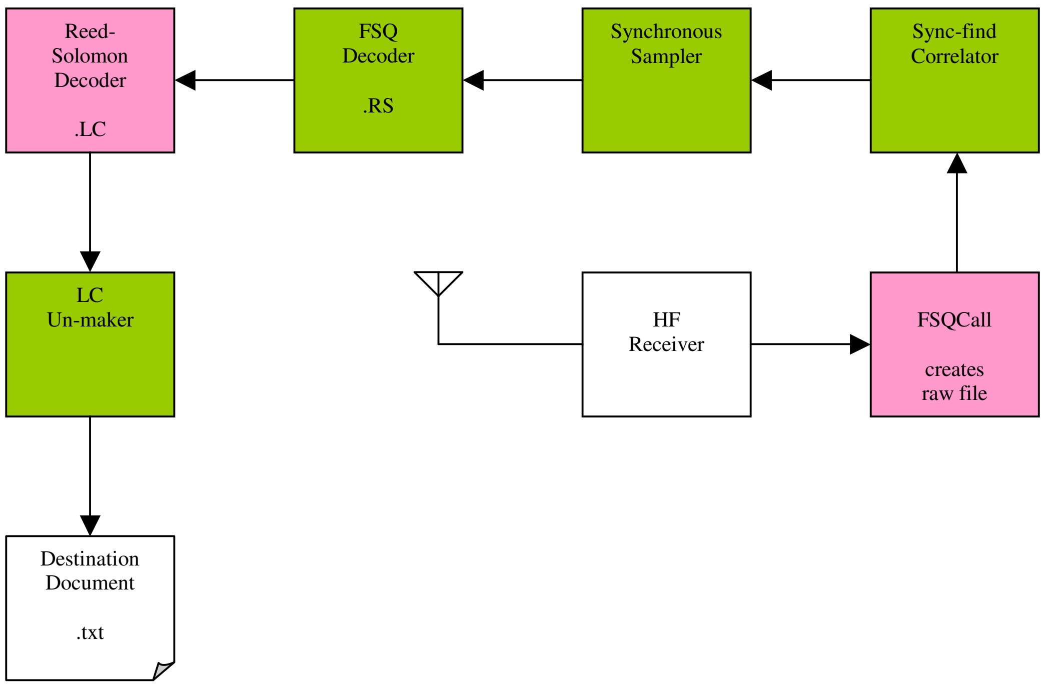
Block Diagrams

Here the transmitter and receiver broad functional sections are shown, as used to send a Forward Error Corrected File.

Transmitter


Source Document .txt
LC Maker .LC
Reed-Solomon Coder .RS
File Transmit Preparation data.txt
FSQCall Sends contents of data.txt
HF Transmitter

Receiver



Developer:

🟪 ZL2AFP

🟩 ZL1BPU