AN2943
Rev. 0, 5/2005

GSM Channel Equalization, Decoding, and SOVA on the MSC8126 Viterbi Coprocessor (VCOP)

By Manoj Bapat, Dov Levenglick, and Odi Dahan

This application note describes the theory and implementation of GSM channel equalization and channel decoding algorithms using the Freescale MSC8126 Viterbi coprocessor (VCOP). It also examines the theory behind the soft output Viterbi algorithm (SOVA) assisted by the VCOP. Code examples illustrate how the VCOP performs channel equalization and the SOVA algorithm. A set of suggested design practices for VCOP usage is followed by a discussion of the VCOP driver, which provides a simple, easy interface to the VCOP. Examples of driver usage cover GSM equalization, channel decoding, and SOVA. The source code and the header file for the VCOP driver are also presented.

CONTENTS

1 Basics of Convolutional Encoding/Decoding

Convolutional coding is a method of transmitting code words consisting of 1/rate symbols affected by data bits instead of transmitting the data bits themselves. Since each data bit is used in more than one code word, the probability of correctly decoding each bit increases. All examples in this document are for an encoder in which rate = 1/2 (two symbols create a code word) and the constraint length is the length of the shift register, *K*, = 3. **Figure 1** shows a pictorial view of such an encoder[1].

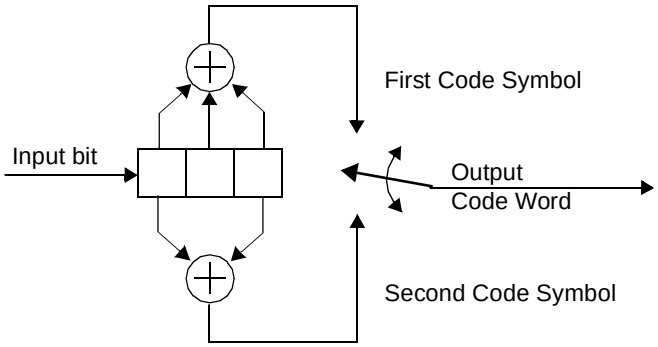

Figure 1. Convolutional Encoder Diagram

The transmitted bits are mapped from {0, 1} to {1, –1}, respectively. The channel adds noise to the transmitted signal, which is generally modeled using additive white gaussian noise (AWGN). This discussion references an example in which the data {1, 0, 0, 1, 1, 0, 0, 0} is transmitted and the last three bits are for flushing the transmitter. **Table 1** lists the expected results.

Table 1. Transitions of the Encoder

State of the Encoder	Input Bit	New State of the Encoder	Transmitted Code Word (First, Second)
000	1	100	11
100	0	010	10
010	0	001	11
001	1	100	11
100	1	110	01
110	0	011	01
011	0	001	11
001	0	000	00

Another view of an encoder is the view of a finite state machine (FSM). **Figure 2** shows the state diagram view of the encoder.

1. The initial state of the encoder is assumed to be all zeros. When all the data is encoded, the encoder is flushed by passing K – 1 zeros through it. The code words that result from these zeros are transmitted as well.

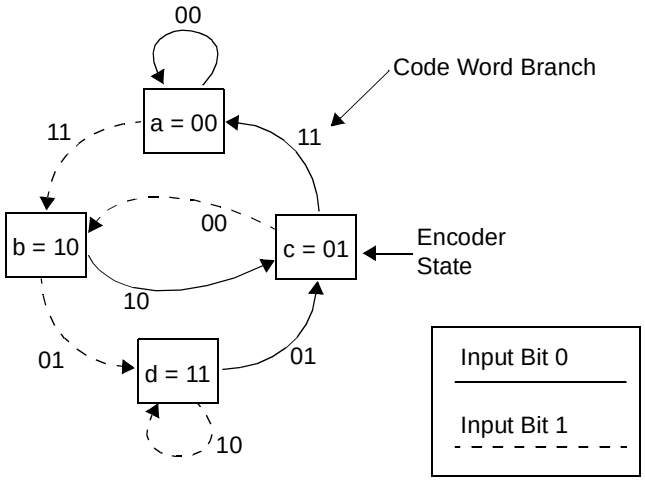

Figure 2. Convolutional Encoder, State Diagram View

Table 2 shows the transitions of the states through transmission.

Table 2. Transitions of the Encoder, State Diagram View

State of the Encoder	Input Bit	New State of the Encoder	Transmitted Code Word (First, Second)
a	1	b	11
b	0	c	10
c	0	a	11
a	1	b	11
b	1	d	01
d	0	c	01
c	0	a	11
a	0	a	00

The rightmost column in **Table 1** is equal to the rightmost column in **Table 2**. Although the state diagram provides all the information needed to follow the transitions of the convolutional encoder in time, this is not an easy task. To simplify tracking of the encoder transitions, we introduce the tree diagram in **Figure 3**. The bold line follows the transmitted bits (from the example) through the tree.

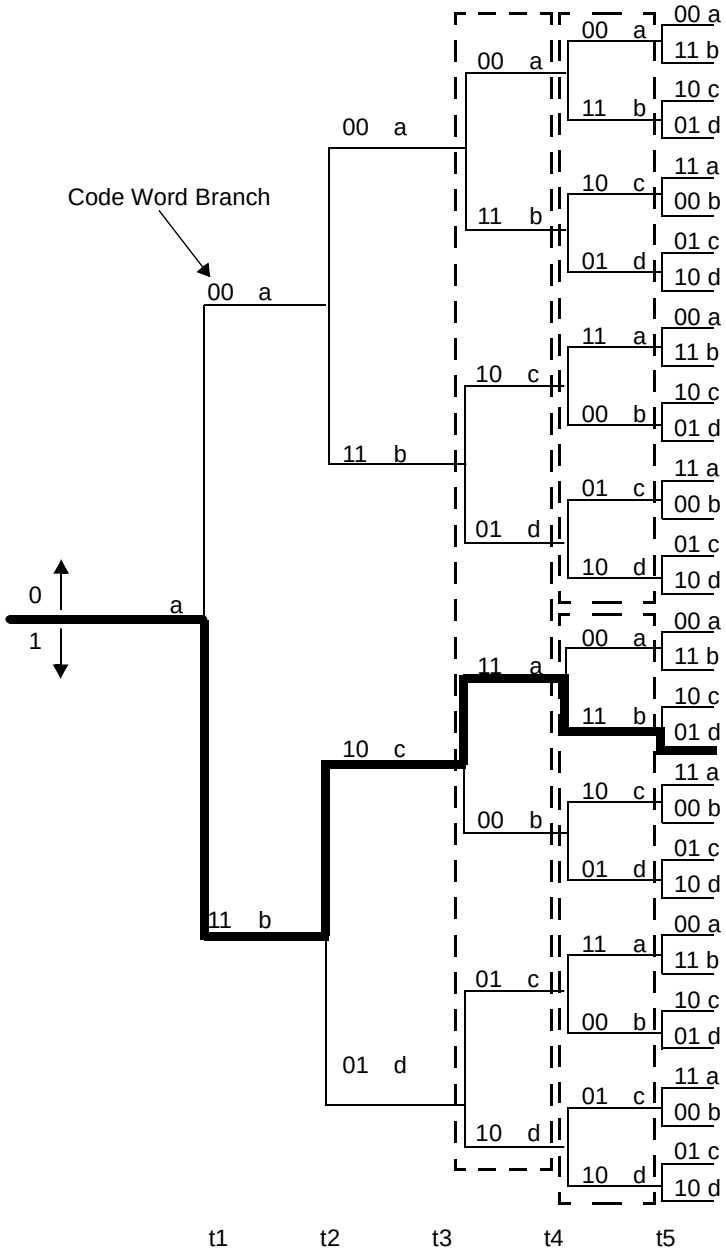

Figure 3. Convolutional Encoder, Tree Diagram View

If we were to sum the states and code words around the bold line into a table, this table would be identical to **Table 2**. Tracing the transitions on the tree diagram is easy because the size of the block grows linearly while the size of the tree grows exponentially. Close inspection of the diagram reveals that the tree structure repeats itself at t_4, as the upper and lower half of the structure at t_4 are equal to the whole structure at t_3. These portions of the diagram are surrounded by a dotted line. Generally, any tree diagram starts repeating itself after *K* branches. It is this repetition that the trellis diagram (**Figure 4**) uses to provide all the information graphically and succinctly. Notice that the upper branch from each state represents a transmitted 0, while the lower branch represents a transmitted 1. As expected, the flow through the trellis diagram results in **Table 2**.

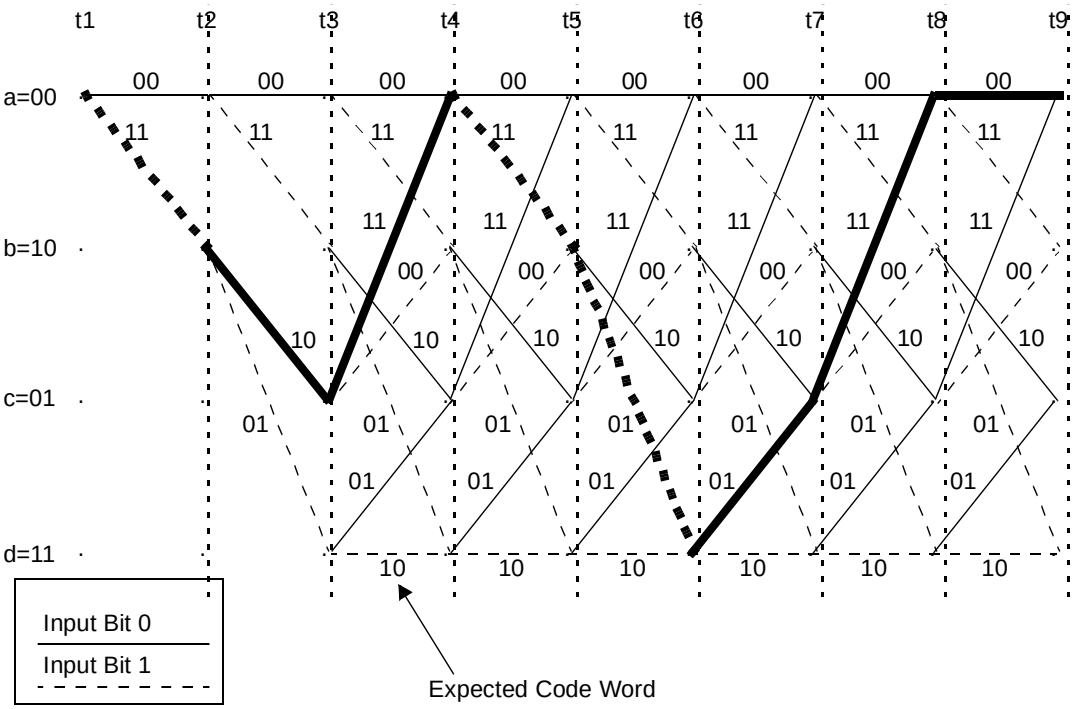

Figure 4. Convolutional Encoder, Trellis Diagram View

1.1 Viterbi Decoding

To decode the received data, we must maximize the likelihood function of the received sequence (**Z**) given all possible transmitted sequences ($\mathbf{U}^{(m)}$). $P(\mathbf{Z}|\mathbf{U}^{(M')}) = \max P(\mathbf{Z}|\mathbf{U}^{(m)})$ over all $\mathbf{U}^{(m)}$. The complexity of calculating the probability and therefore the computational load grows as the length of the received data grows, much as the complexity increases when we trace the received data using the tree diagram (**Figure 3**). To address this problem, Viterbi devised and proved an algorithm to detach the size of the received data from the complexity of calculation. The maximum likelihood function is calculated on each received code word rather than on the whole bulk of received data. The Viterbi algorithm calculates the similarity (distance) of the received symbol at time t_i and all branches entering all states at that time. This distance is the branch metric. Following are ways to calculate the branch metrics when the inputs are soft symbols.

- *Euclidean distance*. The Euclidean method involves straightforward distance calculations. The basis for calculating the distance is the 8-bit, twos complement representation of the maximum values. A received 0 is represented as 127 and a received 1 is represented as –127. Assuming that the received code words are $\{x, y\}$ and we are calculating the distance from state *b* to state *d* $\{0, 1\}$, the distance is calculated as shown in **Equation 1**:

$$BM_{Euc} = \sqrt{(127 - x)^2 - ((-127) - y)^2} \qquad \textbf{Equation 1}$$

- *Manhattan distance*. The Manhattan metric calculates the multiplication of the expected data in the state and the received data. Assuming we are calculating the BM from state *b* to state *c* $\{0, 1\} = \{1, -1\}$ and the received data is $\{x, y\}$, as shown in **Equation 2**:

$$BM_{Man} = (1 \cdot x) + ((-1) \cdot y) \qquad \textbf{Equation 2}$$

The Euclidean distance has proven to be mathematically optimal, but it is extremely difficult to implement in hardware. Although the Manhattan metric is not as mathematically powerful, it has proven to be effective and is easy to implement in hardware. Therefore, the VCOP uses the Manhattan metric, as do many other Viterbi coprocessors on the market.

If the inputs are hard symbols, the distance is calculated as the number of different bits between the expected input and the actual input. At time t_1 all states have a path metric (PM) of 0, the PM being the summation of all the branch metrics (BMs) on the surviving path up to that state. The BM from each two states in stage t_{i-1} that enter the same state in stage t_i are added to their respective PM from each state to create two possibilities for surviving paths. However, only the path with the best metric, the surviving path, is saved as the PM of the state in stage t_i. **Figure 5** shows an example of channel decoding that uses the Viterbi algorithm with hard inputs. The surviving path is the one with the lowest PM. The surviving path to each state is marked in a bold line. The PM numbers are shown in bold.

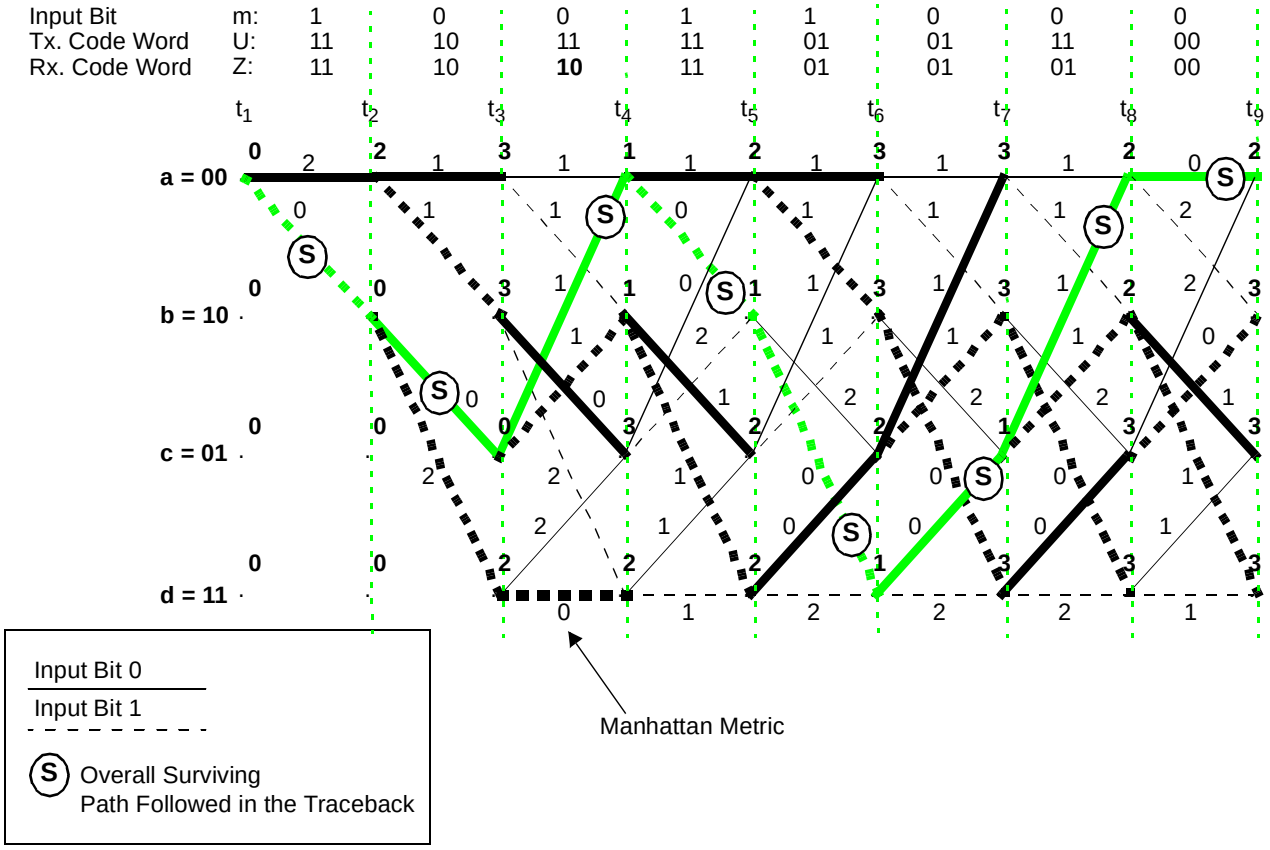

Figure 5. Channel Decoding With Hard Inputs

At t_6, the PM from both states in t_5 into states *a* and *b* add up to 3. The upper path was chosen as the surviving path because the example is consistent with the VCOP. However, both choices are equally right. After all decisions are made about the surviving paths of all states in all stages, we can trace back on the trellis diagram find the transmitted data. The decoded data is {1, 0, 0, 1, 1, 0, 0, 0}. That is, BER = $\frac{0}{8} = 0$), even though two errors were received. At t_9, if there are two states with equal minimal PMs, we have two options:

- Choose either path (the VCOP chooses the upper state).
- Use the prior data at hand, that is, that the encoder was flushed. When this data is taken in to account, it is clear that the path to be chosen is the path ending in state *a*.

Figure 6 shows an example of decoding with soft inputs. The decoded data is {1, 0, 0, 1, 1, 0, 0, 0}. That is, BER = $\frac{0}{8} = 0$), even though two errors were received. The BM are calculated in **Equation 2** for the maximal PM. In **Figure 5**, we use minimum Pm, and in **Figure 6** we use maximum PM because of the different ways of calculating the BM. In the former, the BM measures the number of different bits in the expected and received, and we want a

minimal difference. The latter multiplies the received soft value by the expected value using the maximum values {127, –127}. If the sign of the expected value matches the sign of the received value, the multiplication results in a positive integer. We want the maximum product of these integer summations.

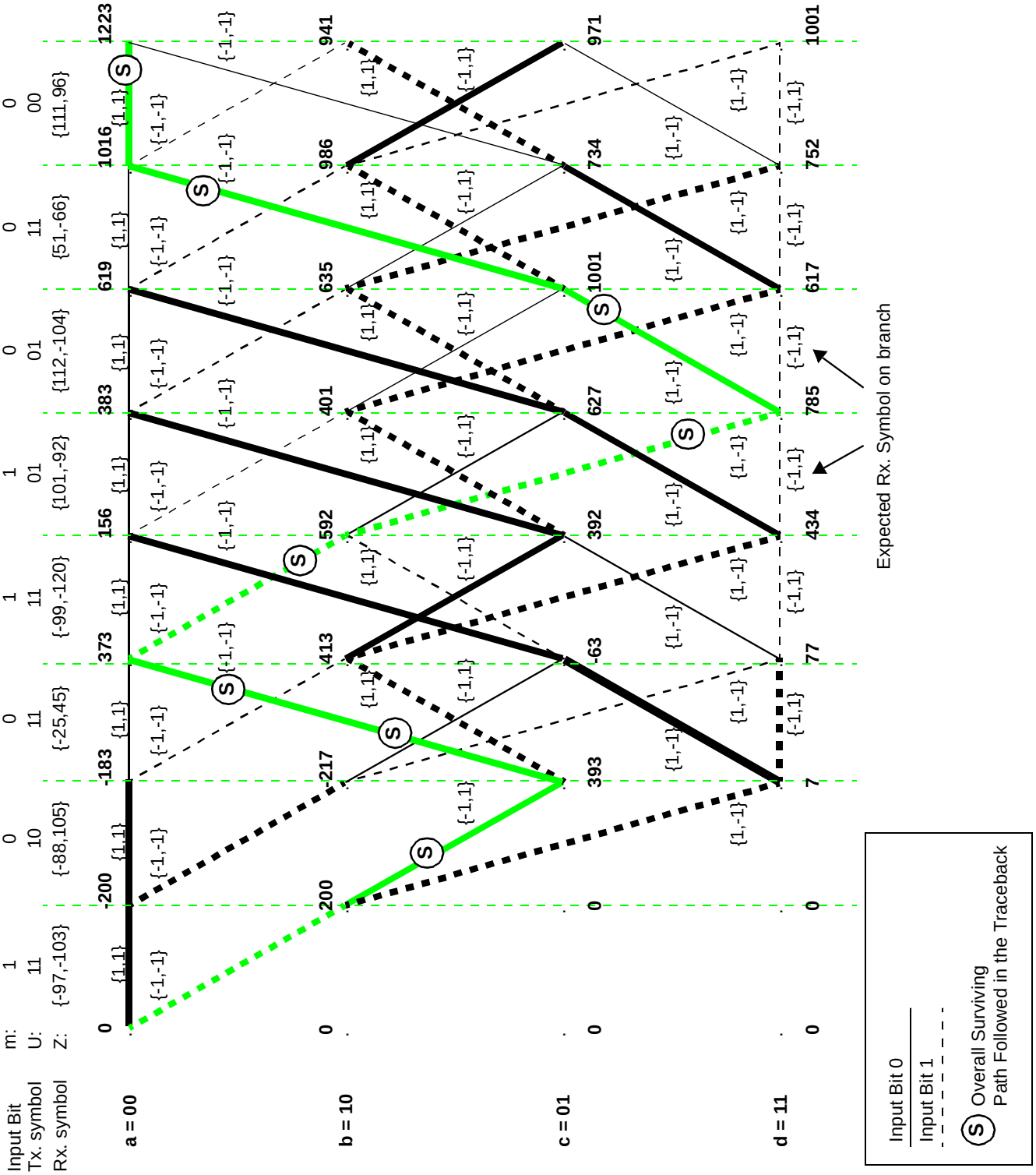

Figure 6. Channel Decoding With Soft Inputs

1.2 GSM Equalization, Channel Decoding and SOVA

The channel coding scheme, as specified by GSM, is depicted in **Figure 7**.

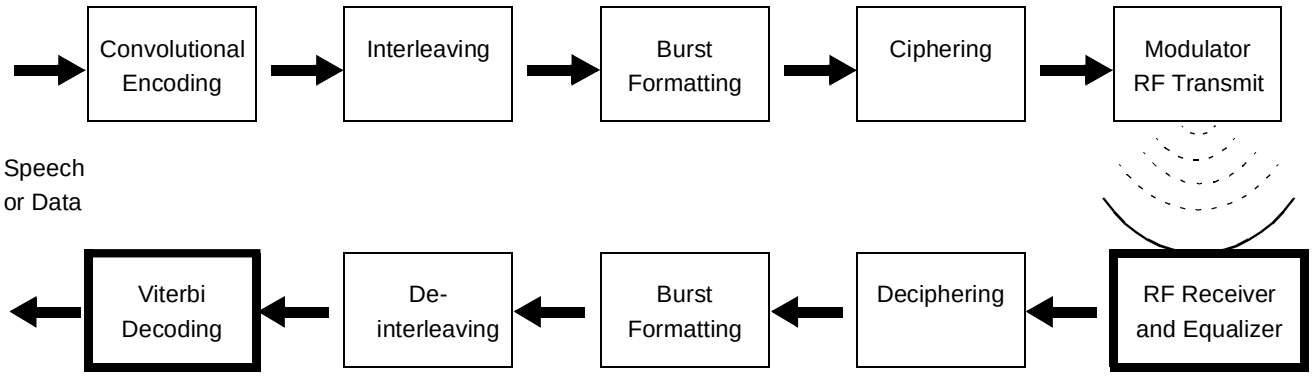

Figure 7. Block diagram of a typical GSM data communication system

The blocks that use the Viterbi algorithm are Viterbi decoding and channel equalization (**bold** framed). The next paragraphs present the theory of channel equalization and a general implementation of VCOP in a GSM equalizer.

1.2.1 Channel Equalization Theory

The channels of time-division multiple access (TDMA) systems such as GSM introduce multi-path interference. The transmitted signal bounces between reflecting obstacles, such as buildings and hills, and reaches the receiver at various delays and attenuation values. In any given time slot, the received signal contains echoes from transmissions made at various times. These echoes are inter-symbol interference (ISI). The channel may therefore be modeled as an FIR filter with impulse response $h_c(t)$. The output of this FIR filter is a sum of its delayed and attenuated inputs, just as the received signal is a sum of delayed and attenuated paths.

In GSM, every transmitted slot includes the data bits and a training sequence, $s_{tr}(t)$, that is also known to the receiver. This training sequence enables the receiver to perform channel estimation and equalization. The received training sequence, $r_{tr}(t)$, is a convolution of the transmitted training sequence, $s_{tr}(t)$, and the channel's impulse response, $h_c(t)$:

Equation 3

$$r_{tr}(t) = s_{tr}(t) \otimes h_c(t)$$

The received training sequence in the digital domain, $r_{tr}[k]$, is fed into a digital matched filter with an impulse response, $h_{mf}[k]$, that is matched to $s_{tr}[k]$. The matched filter output, $h_e[k]$, can be written as shown in **Equation 4**.

Equation 4

$$h_e[k] = r_{tr}[k] \otimes h_{mf}[k] = s_{tr}[k] \otimes h_c[k] \otimes h_{mf}[k] = R_s[k] \otimes h_c[k]$$

where $R_s[k]$ is the auto-correlation function of $S_{tr}[k]$. In GSM, the training sequences are engineered so that $R_s[k]$ is a highly peaked (impulse-like) real function. Therefore, $h_e[k]$ is a good estimation of the complex value $h_c[k]$.

Let L denote the channel memory. That is, the channel has $L+1$ taps. $x[k]$ is the complex valued sample of the received signal at time k. The Viterbi equalizer finds the sequence $a[k] \in \{-1,+1\}$ that minimizes the Euclidian metric in **Equation 5**. Thus, $a[k]$ is the maximum likelihood sequence estimation (MLSE) that is the optimal estimation of the input symbols to the channel (for AWGN channels) [3].

Equation 5

$$\tilde{M}[k] = \sum_{l=0}^{k} \left| x[l] - \sum_{i=0}^{L} h_c[i] \cdot a[l-i] \right|^2 \approx \sum_{l=0}^{k} \left| x[l] - \sum_{i=0}^{L} h_e[i] \cdot a[l-i] \right|^2$$

This Euclidian metric can be mathematically approximated to another metric called the matched filter metric, *M[k]*. Due to a change in sign, we now maximize *M[k]* in **Equation 6**:

Equation 6

$$M[k] = \sum_{l=0}^{k} a[l] \cdot \Re\left(y[l] - \sum_{i=1}^{L} S_i \cdot a[l-i] \right)$$

where

Equation 7

$$y[k] = \sum_{i=0}^{L} h_e^*[i] \cdot x[i+k]$$

is the output of *x[k]* applied to a matched filter with an impulse response $h_e[k]$ and

Equation 8

$$S_k = \sum_{i=0}^{L} h_e^*(i) \cdot h_e(i+k)$$

$$k = 1, 2, \ldots, L$$

is the *k*-th tap of the autocorrelation of the channel impulse response estimation. The real part of the S_k series is called S-Parameters. Since *a[k]* can be only +1 or –1, and since the S-parameters are known, the expression in **Equation 9**:

Equation 9

$$\Re\left(\sum_{i=1}^{L} S_i \cdot a[l-i] \right)$$

can take only one 2^L value. These values are denoted as Viterbi parameters (*VP*). To maximize the matched filter metric *M[k]*, we must try all the possible *a[k]* sequences. The Viterbi algorithm does exactly this very efficiently using 2^L states. In each stage all the possible *VP* values are calculated (one per state). Also, in each stage two branch metrics are calculated: the first is *+(y – VP)* and the second is *–(y – VP)*. Those path metrics are also called Ungerboeck metrics [2]. Each path in the trellis denotes the value of *M[k]* using different *a[k]*. The MLSE path represents the *a[k]* that maximizes *M[k]*.

1.2.2 GSM Channel Decoding

The Viterbi algorithm is also used in the GSM receiver for channel decoding. The theory behind convolutional encoding and channel decoding is explained in **Section 1**, *Basics of Convolutional Encoding/Decoding*, on page 2. The channel decoder performance significantly improves when soft input symbols are used instead of hard symbols. Therefore, the equalizer implements a soft output Viterbi algorithm (SOVA).

1.3 Soft Output Viterbi Algorithm (SOVA) Assist

We use SOVA for a GSM equalizer because a hard output Viterbi equalizer negates the capability of the outer Viterbi decoder to accept soft inputs for performing forward error correction (FEC). On the other hand, a SOVA-based inner Viterbi equalizer generates soft outputs, which in turn form the soft inputs for the outer Viterbi decoder performing FEC. The Viterbi algorithm is modified to deliver not only the most likely path sequence in a finite chain Markov sequence, but also an *a posteriori* probability or a reliability value for each bit. With this reliability indicator, the modified Viterbi algorithm produces soft decisions for use in the outer Viterbi decoder. Thus, the inner Viterbi algorithm accepts and returns soft sample values. The SOVA algorithm improves the signal to noise ratio (SNR) and is implemented with minimal changes to the original Viterbi algorithm.

Hagenauer and Hoeher [8] outline an algorithm that addresses this problem in two consecutive stages:

1. Channel equalization to minimize ISI.
2. Channel decoding, that is, Viterbi decoding.

According to [8], the VCOP must not only output the hard decision but also either an *aposteriori* probability for each bit or a reliability factor. The MSC8126 VCOP meets this requirement with the ability to provide:

- Hard decision bits.
- History buffer.
- Delta values for all states of all stages.
- Recursive traceback hard decisions.

The hard decision bits are the regular traceback hard decision that is later used as a reference for the SOVA assist algorithm.

The delta values are the difference between the PM + BM (branch metric) for each of the two states (stage n – 1) that are used to calculate the PM of a state in stage *n*. The delta values are PMxyz0_0–PMxyz0_1. In **Figure 8**, 0xyz and 1xyz are the states in the trellis diagram from which the butterfly originates; xyz can be any 3-bit binary value (xyz ∈ {000, . . . , 111}. They differ by the first bit, which is not relevant in the following stage. The trellis states xyz0 and xyz1 are the states at which the butterfly ends, and they have the same xyz value as the originating states. PMxyza_b is the PM from state xyz with the prefix b to state xyz with the suffix a. This PM is based on the PM from the previous stage (PM0xyz and PM1xyz).

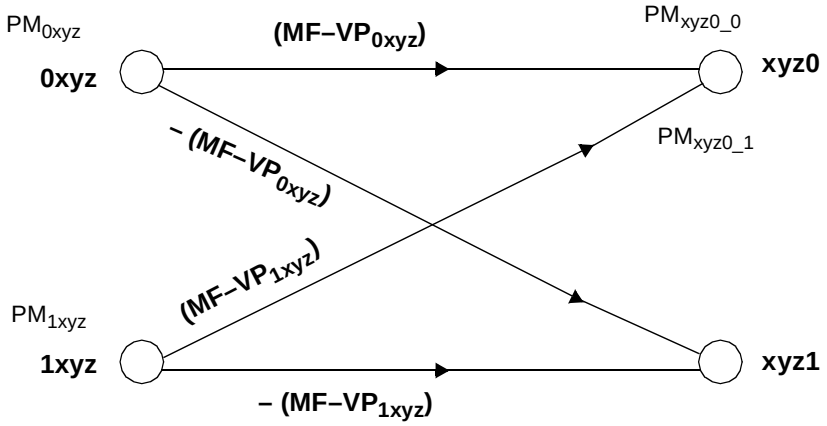

Figure 8. Delta Values

To use the delta values, you can either perform software recursive traceback if VCOP resources are scarce or have the VCOP perform it. The recursive traceback is performed by running over the history buffer, which is why it can be dumped. For each state from N (block size) to 1, a traceback of M stages is performed. The VCOP supports $M \in \{16, 32\}$.

2 GSM Channel Equalization and Decoding on VCOP

The Viterbi algorithm uses the redundant information added to the information bits in the convolutional encoder to perform maximum likelihood decoding on the received data. The main calculation performed in the Viterbi algorithm and on the MSC8126 VCOP is add-compare-select (ACS), as shown in **Figure 9**. We add the BMs to the PMs from the previous stage and compare them to find the best metric, thereby creating the PMs for the current stage. The VCOP performs two butterflies in parallel (see **Figure 10**). The data widths are 16-bit PM, 11-bit BM, and 8-bit soft input symbols.

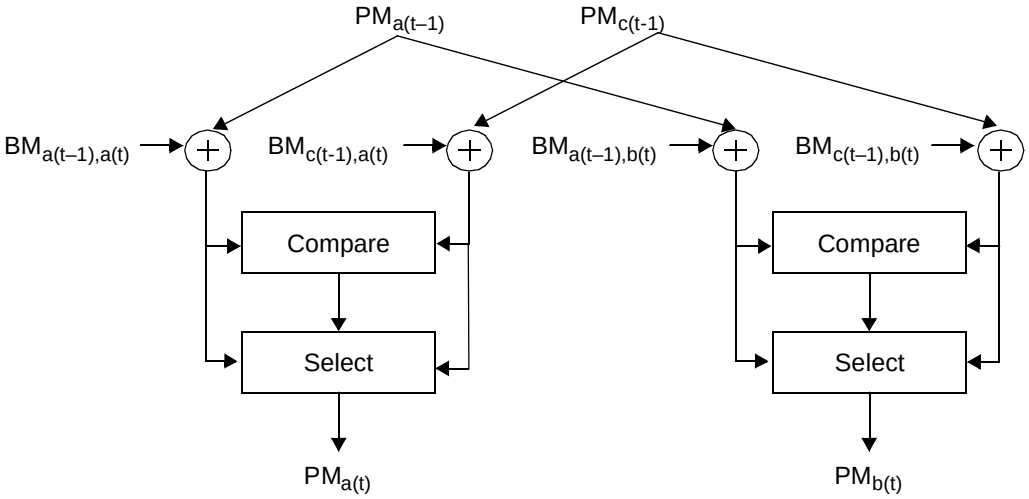

Figure 9. Add Compare Select (ACS)

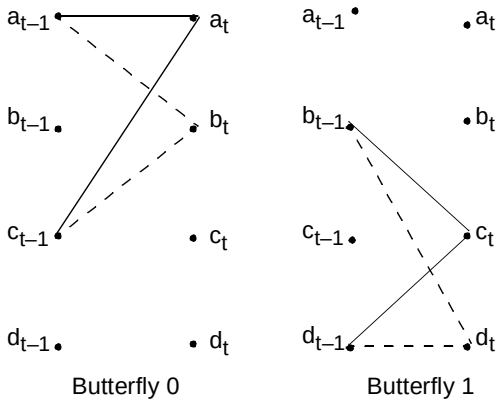

Figure 10. Butterflies

In Equalization mode, the branch metric for each transition within a Viterbi butterfly is a function of the matched filter output (MF) and the Viterbi parameters (VP). The VP values come from the channel autocorrelation. After the channel impulse response coefficients are extracted via a cross correlation process (also referred to as S-parameters estimated channel input response or channel coefficients), the VP value for a particular state, uvwxyz, is calculated as follows:

Equation 10

$$VP(u,v,w,x,y,z) = (-1)^{u} \cdot S_6 + (-1)^{v} \cdot S_5 + (-1)^{w} \cdot S_4 + (-1)^{x} \cdot S_3 + (-1)^{y} \cdot S_2 + (-1)^{z} \cdot S_1$$

Therefore, there can be $2^{\text{Num. of S-Parameters}}$ different VP values for $2^{\text{Num. of S-Parameters}}$ state trellis.This equation is calculated twice, in parallel, every clock to generate 2 VP values. The 2 VPs are used to calculate a butterfly once in every clock cycle. CNST is the constraint length, which defines the relation between the low and high states that compose the butterfly.

Figure 11 shows a typical GSM equalizer with partitioning between the VCOP and the SC140 core functions. The equalizer extracts the training sequence, $r_{tr}[k]$, from the received signal and filters it using a matched filter with impulse response $h_{mf}[k]$. The output of the matched filter, the channel estimation, $h_e[k]$, is autocorrelated to obtain the S-parameters. The S-parameters are then correlated with all the possible 2^L waveforms to generate the *VP*.

The data symbols of the received signal, $x[k]$, are also filtered by a matched filter with impulse response $h_e[k]$. The real part of the matched filter output, $\Re(y[k])$, also denoted as *MF*, is the input for the VCOP.

The branch metric calculator calculates the Ungerboeck metric, which the Viterbi Algorithm uses to find the MLSE. The equalized signal is the MLSE represented as hard decisions (±1).

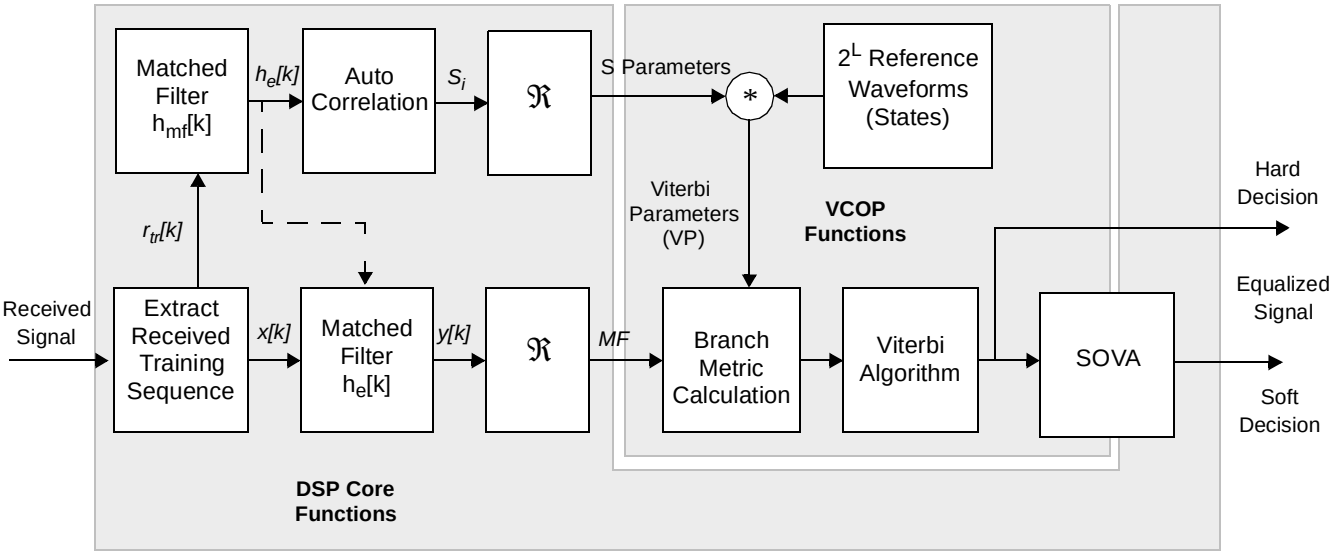

Figure 11. Viterbi Equalizer Application

The Viterbi algorithm is used in the GSM receiver for channel decoding. The SOVA algorithm is composed of three tasks. The VCOP always performs the first task. Either the VCOP or the SC140 core can perform the second task, depending on how the user partitions the tasks. The SC140 core always performs the third task:

1. Delta values calculation. Calculate the difference (delta) between the path metrics that are in-bound to the same state. This task is performed for each state on each stage during the feed forward.

2. Recursive traceback. Generate a concurrent path starting from each state on the MLSE path (recursive traceback). See **Figure 13**.
3. Reliability values update, as follows:
   a. Set the reliability value of all the stages to infinity (or the highest possible positive value).
   b. Compare the hard decisions of each concurrent path to the MLSE path. If the hard decision obtained by the concurrent path on a certain stage differs from that of the MLSE path, set the reliability value of that stage to the lowest value between the current reliability value and the delta value of the starting stage of the concurrent path.
   c. Calculate the soft output symbols by multiplying each hard decision by its corresponding reliability value.

Table 3 lists the maximum number of stages that the VCOP supports for equalization.

Table 3. VCOP Maximum Stages in Equalization

Number of S-Parameters	Number of Stages
4	4090
5	4090
6	3072

There are four types of configuration registers for programming the VCOP:

- *Address registers.* Point to locations in the memory visible to the VCOP from where the input data is read, the output data is written, and the various algorithm assist dumps are performed.
  - VCOP Output Buffer Address Register (VOBAR)
  - VCOP Input Buffer Start Address Register (VIBSAR)
  - VCOP PM Fill Address Register (VPMFAR)
  - VCOP Algorithm Assist Data Dump Address Register (VAADAR)
- *Encoder profiling registers.* Give the VCOP all information necessary to decode the received data: length of the block, K, rate, polynomials, and so on. The reasoning and min/max values for each field are listed in the *MSC8126 Reference Manual*. See **Table 4**.
  - VCOP Configuration Register (VCONFR)
  - VCOP Polynomials Register (VPOLR)
  - VCOP Puncture Pattern Register (VPPR)
  - VCOP PM Init State Register (VPMISR)
- *Session profiling registers.* Give the VCOP the size of the input block and the type of traceback required. See **Table 5**.
  - VCOP Input Buffer Block Length Register (VIBBLR)
  - VCOP Trellis Count Register (VCNT)
- *Algorithm assist registers.* Can be programmed via the VCONFR[ISRNUM] bit to make the VCOP dump the PMs at various stages.
  - VCOP Interim Stage Register (and S-Parameters) A (VISRA). A dual-mode register. For feed-forward channel decoding mode, it holds the stages of PM memory content dumped out of the

VCOP. For feed-forward channel equalization mode, it holds the channel autocorrelation parameters. This register can be read any time but is written only when the VCOP is in idle mode.

{$SPARAM_i$_H,$SPARAM_i$_L} — S Parameter i. Defines the i-th channel coefficient to be used in an equalization feed-forward session. The S-Parameter can be any 16-bit signed value (2^{15}–1 < S Parameter i < –2^{15}).

— VCOP Interim Stage Register (and S-Parameters) B (VISRB). A dual-mode register. For feed-forward channel decoding mode, it holds the stages of PM memory content dumped out of the VCOP. For feed-forward channel equalization mode, it holds the channel autocorrelation parameters. This register can be read any time but is written only when the VCOP is in idle mode.

{$SPARAM_i$_H,$SPARAM_i$_L} — S Parameter i. Defines the i-th channel coefficient to be used in an equalization feed-forward session. The S-Parameter can be any 16-bit signed value (2^{15}–1 < S Parameter i < –2^{15}).

— VCOP Interim Stage Register (and S-Parameters) C (VISRC). A dual mode register. For feed-forward channel decoding mode, it holds the stages of PM memory content dumped out of the VCOP. For feed-forward channel equalization mode, it holds the channel autocorrelation parameters. This register can be read at time but is written only when the VCOP is in idle mode.

{$SPARAM_i$_H,$SPARAM_i$_L} — S Parameter i. Defines the i-th channel coefficient to be used in equalization feed-forward session. The S-Parameter may be any 16-bit signed value (2^{15}–1 < S Parameter i < –2^{15}).

Table 4. VCOP Encoder Profiling Registers

Register	Field	Min Value	Max Value	Description
VPOLR	G0[7–0]	N/A	N/A	Polynomial 0 for all values of VCONFR[RATE] and Polynomial 3, where VCONFR[RATE] = 0x3.
	G1[7–0]	N/A	N/A	Polynomial 1 for all values of VCONFR[RATE] and Polynomial 4, where VCONFR[RATE] = 0x3.
	G2[7–0]	N/A	N/A	Polynomial 2 where VCONFR[RATE] = {0x0–0x2}) and Polynomial 5, where VCONFR[RATE] = 0x3.
	G3[7–0]	N/A	N/A	Polynomial 3, where VCONFR[RATE] = 0x2.
VPMISR	STATE_INDEX[7–0]	0x00	2^{k-1}	After the PM memory is cleared, as it usually is, all states have an initial PM of 0. However, you may already know the state at which the convolutional encoder started coding (in 3G applications the encoder always starts at 0). To give the path starting from that state a better chance of being the surviving path with the best metric after the feed forward, you can program a value other than 0. For minimum PM calculations, program the favored state with a value lower than 0. For maximum PM calculations, program the favored state with a value higher than 0. VPMISR[PM_DATA] is a twos complement number.
	PM_DATA[15–0]	0x3FFF	0xC000	
VCONFR	CMD[5–0]	N/A	N/A	Specifies which sequence of operations to run
	RATE[1–0	0x0	0x3	Specifies the rate at which data is encoded.
	FF_MIN/MAX	0x0	0x1	Specifies the best metric. The standards enable both minimum and maximum BMs to be considered as best.
	CONST[2–0]	0x0	0x4	Defines K between 5 and 9 as specified in the various standards (K = VCONFR[CONST] – 5).
	ISRNUM[3–0]	0x0	0xC	Specifies how many stages to dump for BTFD applications. The VISRx specifies which stages to dump.

Table 5. VCOP Session Profiling Registers

Register	Field	Min Value	Max Value	Description
VCNT	TBNDX[11–0]	K+1	0xFFA[1]	For manually programming the starting stage of the traceback.
	TBMOD[2–0]	0x0	0x3[1]	Specifies whether to use VCNT[LEARN] and VCNT[PDTBS].
	LEARN[5–0]	0x00	0x3F	Not applicable for GSM.
	PDTBS	0x00	2^{k-1}	Enables you not to use the absolute Min/Max (dependant on VCONFR[FF_MIN/MAX]) PM but rather to specify the starting state of the traceback. This is useful when you know the state of the encoder when it finished coding the data and therefore the location where the decoder should start decoding.
VIBBLR	VIBBL[13–0]	0x0001[1]	0x1FF4	The maximum value is the multiplication of the maximum block size at Rate = 1/6 (2730) and 6.

NOTES: 1. The maximum value depends on VCONFR[RATE] and VCONFR[CONST] and can be found per each combination in the VCOP chapter of the *MSC8126 Reference Manual*. The value is the overall maximum value.
2. The values 0x4–0x7 are used in channel equalization.
3. Although 0x0001 is a legal value for VIBBLR[VIBBL], the minimum legal value is K + 1 to perform traceback sessions.

After the VCOP is programmed, it starts processing the data with no intervention from the SC140 core. As it completes each stage in the process (not each stage in the trellis), the VCOP can interrupt the SC140 core. The overall stages of VCOP operation are summarized as follows:

1. Initialize PM memory using the PM FILL or PM CLEAR command and occasionally VPMISR.
2. Read the data required for each stage in the trellis diagram and process the data.
3. Dump the history memory in accordance with VCONFR[CMD].
4. Perform the traceback on the history memory in accordance with VCONFR[CMD].

3 Recommended Design Practices

This section presents the VCOP features and recommends ways you can get the most benefit from these features.

3.1 VCOP Features

- Fully programmable feed-forward channel decoding and traceback sessions.
- Channel decoding:
  - Constraint length between K = 5 and K = 9.
  - Puncture codes.
  - Rate of 1/2, 1/3, 1/4, and 1/6.
  - Four fully programmable polynomials (1/6 rate is implemented by three polynomials only).
- The input to equalization is 16-bit. The input symbols for decoding are 8-bit (256 levels) signed soft symbols.
- Output is hard decision (1-bit).

- Fully programmable block length for all sessions.
- Programmable traceback methods of maximum PM, minimum PM, or end state.
- Start of feed-forward according to a pre-saved PM memory content. However, the history buffer is not saved, so the traceback is according to the current block only.
- SC140 core programs the VCOP parameters while the VCOP is in idle mode and then the VCOP can run independently on the whole block of data.
- The number of users the VCOP supports is linear to frequency.
- Fully programmable feed forward channel equalization and traceback sessions.
- For GSM channel equalization:
  - Fully programmable 4 to 6 estimated channel autocorrelation coefficients (S-parameters).
  - History buffer with up to 4090 stages for GSM.
  - Matched filter input is 8 bits (256 levels).
  - SOVA assist algorithm.
  - Output 8-bit coded delta values for SOVA assist algorithm, 1-bit hard decision traceback, and history buffer or recursive traceback.
- Input symbols are 8-bit (256 levels) signed soft symbols.
- Output is hard decision (1-bit).
- Fully programmable block length for all sessions.
- Programmable traceback methods of Max path, Min path, or end state.
- Programmable learning period length for the traceback session.
- Start of feed forward according to a pre-saved PM memory content. However the history buffer is not saved. Therefore the traceback is according to the current block only.
- SC140 core programs the VCOP parameters while the VCOP is idle and then the VCOP independently processes the block of data.
- Interrupt lines and status bits notify the SC140 core when a session completes.
- Memory accesses are multiples of 8 bytes, even if meaningful data holds a fraction of 8 bytes, with a value of 0 in the filler bits.
- Performance monitoring unit with six monitored behaviors.

3.2 Interrupts Versus Polling

Although the VCOP allows polling, it is optimized for an interrupt-driven system. If the SC140 core is busy polling, it either constantly polls the VSTR, thus loading the bus and rendering at least one AGU and one DALU useless to the other processes, or it polls the VSTR occasionally, thus not catching the exact moment at which the VCOP is free for a new configuration. Either of these polling activities degrades performance. Moreover, during BTFD, it is not enough to poll the VSTR. The VSTR[PMD_DONE] bit must be cleared after each PM dump to ensure that the next PM dump is performed. Constantly clearing this bit calls for constant polling.

The disadvantage of an interrupt-driven approach is that each interrupt costs approximately 100 core clock cycles[2]. However, the reasons against polling outweigh the one for polling, so we recommend the use of interrupts. You can use a combination of interrupts and polling so that when the SC140 core initiates a VCOP session, it starts polling the VSTR when the session is supposed to finish. The expected finish time is known because the VCOP needs approximately 67 cycles per symbol, and the number of symbols is known. When the SC140 core detects that the

2. Based on the SmartDSP OS for StarCore-based architecture.

VCOP has finished the session, it starts a new session in a single-core operation or passes control of the VCOP to a different core in a mult-core operation.

3.3 Buffer Allocation

There is no time for you and/or the VCOP driver to allocate buffers in a real-time system, so you must allocate enough space for the four types of buffers the VCOP may use:

- Input buffer, to which the VIBSAR points
- Output buffer, to which the VOBAR points
- Algorithm assist buffer, to which the VAADAR points
- PM fill buffer, to which the VPMFAR points

When it is called, the VCOP driver provided as an example assumes that the code always points to the correct segment of each buffer.

3.4 Single-Core Operation

For single-core operation of the VCOP, you have all the information on a session and can simply program the VCOP by writing directly to the VCOP registers or by using the driver described in **Section 4**.

3.5 Multi-Core Operation

For multi-core operation of the VCOP, following are ways to share the common VCOP resource:

- *Use each SC140 core to perform part of the process* necessary for GSM channel equalization and decoding, with only one SC140 core actually using the VCOP. This is a type of single-core operation, so all information in **Section 3.4** applies—with the difference that the SC140 core activating the VCOP is exempt from all decoding calculations before and after VCOP operation. The advantage of this approach is that only one of the four SC140 cores accesses the VCOP, thus rendering the whole resource management mechanism redundant. The disadvantage is that the load on the SC140 cores is probably not symmetrical because activating the VCOP is a relatively quick and easy task.
- *Use hardware semaphores*. The advantage of this approach is that it uses the hardware and requires minimal programming overhead. The disadvantage is that aborting the current VCOP operation is not straightforward because only the SC140 core that locked the semaphore can unlock it.
- *Create a software semaphore that instantiates a token ring*, or round-robin mechanism (see **Figure 12**) in which each SC140 core controls the VCOP an equal amount of time. The advantage of this approach is that it is easy to implement and requires little overhead. However, it is unlikely that all cores receive the same amount of data to process. If one SC140 core always receives smaller channels, it controls the VCOP a disproportionately small amount of time and may run out of data to process. The token ring/round robin mechanism puts a priority scheme in place. An SC140 core can raise its priority, thus increasing the probability of receiving control over the VCOP.

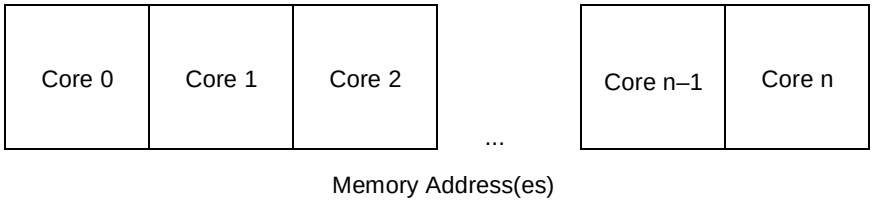

Figure 12. Token Ring for Multiple Cores

- *Create a software semaphore in which the next SC140 core to control the VCOP is the one that has processed the fewest incoming symbols.* This approach ensures that each SC140 core controls the VCOP for the about same amount of time. The disadvantage is that the overhead of calculating the next SC140 core to control the VCOP consumes much more time than a simple token ring. The next SC140 core to control the VCOP is determined according to **Equation 11**, where *N* is the number of cores on the DSP device.The token ring option should be implemented if the SC140 cores handle the same amount of data (on the average), making the more complicated method redundant.

Equation 11

$$min\left(\frac{Processed\ Bits_i}{\sum_{n=0}^{N-1} Processed\ Bits_n}\right)$$

3.5.1 Requesting and Releasing VCOP Services

An SC140 core requests VCOP services as follows:

1. Use a read-modify-write on the address where the queue is managed to find whether the VCOP is in use. Then set the bit that signals to the other SC140 cores that the VCOP is in use.

   If no other SC140 core is using the VCOP, the requesting core can use it. If the VCOP is in use, the requesting SC140 core waits until the VCOP is free.

2. When an SC140 core finishes using the VCOP, it issues a read-modify-write on the address where the queue is managed. The read establishes which SC140 core is next in line to use the VCOP (generally the SC140 core to the right of the one freeing the resource). The write clears the SC140 core bit in the queue. Then it initiates a virtual interrupt (via the GIC) to the SC140 core next in line to use the VCOP.

If the SC140 core needs further VCOP services, it can perform a simple read since clearing the bit is not necessary.

3.5.2 Aborting VCOP Sessions

To abort a VCOP session, the SC140 core currently using the VCOP must enable the VCOP abort interrupt in the LIC and PIC. An SC140 core aborts the current VCOP session (due to a high priority channel), as follows:

1. Enable the LIC and PIC to have the VCOP interrupt.
2. Write a value of 1 to VCONFR[ABORT] and wait for the resulting interrupt.
3. Initiate a VCOP session. When the session terminates, the token ring continues from the interrupting SC140 core.
4. Ignore any data that may have already come from the current session.
5. Disable VCOP interrupts via the LIC and PIC.

3.6 GSM Channel Equalization

The steps in GSM channel equalization are as follows:

1. Clear PM memory or load the initial state from external memory using the PM FILL command.
2. In the VPMISR, initiate any number of states to a desired starting value. Alternatively, configure VPMISR to initiate a single state after an automatic feed forward session, which clears the PM memory and initiates it according to the configuration of VPMISR. In the automatic approach, only a single state can be initialized and only the higher or lower part of it (not both).
3. Update the VIBSAR with the starting address of the raw data to be equalized, and in the VIBBAR, specify the amount of data.
4. Program the channel autocorrelation parameters (S-parameters) into the VISRA, VISRB, and VISRC.
5. Store the output address for the delta values of the SOVA assist, in VAADAR.
6. If automatic traceback is to be used, update the address for storing the output of the traceback session in VOBAR.
7. Start the session using one of these methods:
   - Start feed-forward channel equalization session command.
   - Start feed-forward channel equalization session and start HD traceback session automatically.
   - PM CLEAR and automatically start a feed-forward channel equalization session, followed by an HD traceback session in the VCONFR, together with the necessary parameters (such as rate, puncture code, constraint length, and interrupts to the SC140 core).

3.6.1 Recursive Traceback for SOVA Assist

In equalization mode, the SOVA requires special tracing back from each point on the regular traceback result. Each point is a root for traceback, but the first step is opposite to the one in the original traceback. It is not necessary to trace back all the way, since the branch normally combines with the original traceback after a few stages. The VCOP handles 16-stage or 32-stage recursive traceback. These data sizes are convenient because of the bus width, and they hold sufficient information. **Figure 13** shows an example of recursive traceback. The traceback starting-point is specified by the VCONFR[MIN, MAX] bits or the PDTBS field. The traceback uses the learning period.

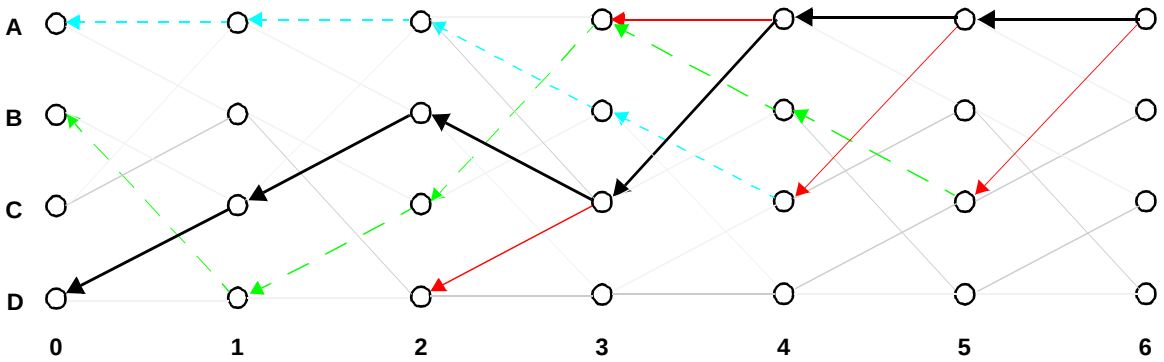

Figure 13. Recursive Traceback

3.7 GSM Channel Decoding

The steps in GSM channel decoding proceed as follows:

1. Clear PM memory or use the PM FILL command to load the initial state from external memory.
2. In the VPMISR, initiate any number of states to a desired starting value.

   Alternatively, configure the VPMISR to initiate a single state after an automatic feed-forward session, which clears the PM memory and initiates it according to the configuration of VPMISR. In the automatic method, only a single state can be initialized.
3. Update the VIBSAR with the starting address of the equalized data to be decoded and the VIBBAR with the amount of data.
4. Program the polynomials into VPOLR.
5. If automatic traceback is to be used, update the address for storing the output of the traceback session in VOBAR.
6. Start the session by specifying the session type in the VCONFR (see **Table 6**), together with the necessary parameters, such as rate, puncture code, and constraint length. Enable the necessary interrupts to the SC140 core.

Table 6. VCOP Equalization Session Types

Type	Description
Equalization feed-forward	The VCOP enters this mode when the “equalization feed-forward” command is written in Idle mode. The SC140 core configures the parameters for this operation, such as rate, S-Parameters, block length, and so on, before it issues this command. The VCOP can return to Idle mode if the ABORT command is issued. When the session terminates as specified in the VCNT register, the mode can change to Traceback if automatic traceback is enabled or return to Idle mode.
History buffer dump	The contents of the history buffer are part of the data required by the DSP code to perform the equalization. Therefore, a history buffer dump occurs automatically unless a recursive traceback is performed. A command to the VCOP can also cause a buffer dump when the decoding process is debugged. For an explicit command to the VCOP, the block size and the constraint length must be programmed correctly.
Hard decision traceback	The VCOP uses the traceback engine and the history buffer memory for a traceback after feed-forward decoding or feed-forward equalization. The bits are packed to 64-bit words. The SC140 core should program the parameters of the traceback operation in advance in Idle mode.
Equalization feed-forward with automatic traceback	A combination of the regular equalization feed-forward session and the traceback session. The main feature is the automatic change of session from feed-forward to traceback.
PM memory clear with automatic equalization feed-forward	A combination of the PM memory clear and regular equalization feed-forward session. The main feature is the automatic change of session from the PM memory clear to feed-forward.
PM memory clear with automatic equalization feed-forward and traceback	A combination of the PM memory clear, equalization feed-forward session and the hard decision traceback session.

4 VCOP Driver

The VCOP driver is a simple, easy interface to the VCOP. This section covers both the driver structs and driver functions. It also provides the driver header file and source code.

4.1 Driver Structs

The VCOP driver structs are listed as follows:

- *vcopStruct*. The driver view of the VCOP and the session, which includes the references in **Table 7**.

Table 7. vcopStruct Internal References

Type	Name	Description
VCOP	vcop_regs	A pointer to the VCOP base address.
vcopSessionStruct	session	A pointer to the current session.
vcopBTFDManagement	btfd_management	A data structure for BTFD

- *VCOP*. Points to the addresses of all VCOP registers.
- *vcopSessionStruct*. Points to all the data to run the current session, including the interim dump points for calculating $S(n_{end})$ and the CRC used during encoding (see **Table 8**).

Table 8. vcopSessionStruct Internal References

Type	Name	Description
vcopAddressPointers	pointers	A pointer to a struct of addresses.
vcopFFStruct	ffData	A pointer to the relevant vcopFFStruct that contains data on the constraint length, polynomials, number of interim dumps (all relevant for programming VCONFR), and the CRC.
vcopTBStruct	tbData	A pointer to the relevant vcopTBStruct that contains data on the size of the input block and the type of traceback.
vcopInterimStruct	visrxData	An array defining the stages in the decoding where an interim dump is performed

- *vcopBTFDManagement*. Holds all information required for running BTFD. The most relevant part of this struct is the array stage_data, which contains a 12-element array of the vcopInterimStage struct. See **Table 9**.

Table 9. vcopInterimStage internal references

Type	Name	Description
bool	performed_traceback	Notifies the driver of whether a traceback has been performed at this stage.
bool	passed_CRC	Notifies the driver of whether the traceback result passed CRC.
fp32	Sn	The value of Sn (see Equation. 1 in Section A.1.2 of 3GPP TS 25.212).
uint32	vobar	A pointer to the traceback resulting from this interim dump.

For all other structs, refer to the VCOP driver header file in **Section 4.3**, *Driver Header File,* on page 29.

4.2 Driver Functions

This section presents the following driver functions:

- Activate the VCOP, vcop_loctl, **Section 4.2.1**, *Activate the VCOP, vcop_Ioctl*, on page 22.
- Write the session configuration to the VCOP, **Section 4.2.2**, *Write Session Configuration to VCOP, vcop_CreateConfig*, on page 26.
- Initialize the VCOP driver, **Section 4.2.3**, *Initialize the Driver, vcop_InitializeDriver*, on page 28.

4.2.1 Activate the VCOP, vcop_Ioctl

```
        #include "vcopDriver.h"
        extern volatile vcopStruct *global_dev_ch;
        vcopStatus vcop_Ioctl(void *dev_ch, uint8 ioctl, void *param)
{
                volatile vcopStruct          *vcop            = (vcopStruct*)dev_ch;
                volatile vcopSessionStruct   *session_params  = (vcopSessionStruct*)param;
                volatile vcopBTFDManagement  *btfd_management = &vcop->btfd_management;
                volatile vcopAddressPointers *pointers        = session_params->pointers;
                volatile VCOP                *vcop_regs       = vcop->vcop_regs;
                // VCONFR[9:2]
                uint8 interrupt_mask = 0x00;
                // flag indicating if history dump is performed
                bool history_dump =   FALSE;
                // flag indicating if traceback is performed
                bool traceback =      FALSE;
                // flag indicating if interim stages are dumped - for BTFD
                bool interim_stages = FALSE;
                uint8 i;
                global_dev_ch = vcop;
                vcop->session = session_params;
/*****************************************************************************/
/*                              Abort                                        */
/*****************************************************************************/
                if (ioctl == 0x00)
                {
                        ENABLE_VCOP_INTERRUPTS_LIC();
                        WRITE_UINT32(vcop_regs->VCONFR, 0x00000005);
                        return VCOP_SUCCESS;
                } // if (ioctl == 0x00)
/*****************************************************************************/
/*                            LPU                                            */
/*****************************************************************************/
                if (session_params->updateLpu)
                {
                        if(session_params->lpuData->reset)
                        WRITE_UINT32(vcop_regs->VPCR, 0x00000000);
                        vcop_createConfig(configLPU, dev_ch);
                } // if (session_params->updateLpu)
                switch (ioctl)
                {
/*****************************************************************************/
/*                            Illegal Configurations                         */
/*****************************************************************************/
```

```
                    case 0x11:
                    case 0x12:
                    case 0x13:
                    case 0x21:
                    case 0x22:
                    case 0x23:
                    case 0x31:
                    case 0x32:
                    case 0x33:
                            return ILLEGAL_CONFIG;
/*****************************************************************************/
/*                          "Real" sessions                                  */
/*****************************************************************************/
                    default:
                            ENABLE_VCOP_INTERRUPTS_LIC();
                            // if there is need to update VPOLR/VPPR/VPMISR/VCONFR
                            if (session_params->updateFF)
                                vcop_createConfig(configFF, (vcopStruct*)vcop);
                            // if there is need to update VIBBLR/VCNT
                            if (session_params->updateTb)
                                vcop_createConfig(configTB, (vcopStruct*)vcop);
                            // mask VCONFR:CMD and all the interrupt enables
                            session_params->ffData->vconfr &= 0x00FFFE00;
/*****************************************************************************/
/*                          Decoding                                         */
/*****************************************************************************/
                            if ((ioctl & 0x0C) == 0x04)
                            {
                                if (session_params->updateVisrx)
                                     vcop_createConfig(configBTFD, (vcopStruct*)vcop);
                    /*************************************************************/
                    /* Analyze ioctl and decide whether there are interim dumps,  */
                    /* history dumps, and/or traceback                              */
                    /*                                                           */
                    /* IMPORTANT: Interim dumps mask tracebacks and history dumps */
                    /*************************************************************/
                                if (session_params->ffData->interimDumps)
                                {
                                     interim_stages = TRUE;
                                     ioctl = ioctl & 0xFC;
                                }
                                else
                                {
                                     if (ioctl & 0x01)
                                          traceback = TRUE;
                                     if (ioctl & 0x02)
                                          history_dump = TRUE;
                                }
                    /*************************************************************/
                    /* Update the BTFD management struct in dev_ch with the       */
                    /* relevant data to this session                             */
                    /*************************************************************/
                                if (interim_stages)
                                {
                                // See Section 4.3, Driver Header File, on page 29
                                #ifdef VCOP_BTFD_PRE_KNOWLEDGE
                                btfd_management->PM_dump_size = (uint16)VCOP_BTFD_PRE_KNOWLEDGE;
                                #else
                                btfd_management->PM_dump_size =
                                (uint16)(1 << session_params->ffData->constraintLength);
```

```
                                #endif
                                btfd_management->num_of_remaining_BTFD_tracebacks =
                                session_params->ffData->interimDumps;
                                WRITE_UINT32(vcop_regs->VAADAR, (uint32)pointers->vaadar);
                                     for (i=0 ; i< session_params->ffData->interimDumps;
                                     i++)
                                     {
                                       btfd_management->stage_data[i].performed_traceback
                                       = FALSE;
                                       btfd_management->stage_data[i].passed_CRC = FALSE;
                                       btfd_management->stage_data[i].Sn = (fp32)-1;
                                     }
                                     btfd_management->vobar = (uint32)pointers->vobar;
                                } // if (interim_stages)
                           /*************************************************************/
                           /* Calculate the next interrupt and enable it                */
                           /* Only one interrupt is enabled according to this priority:  */
                           /* ipi_int_tb, ipi_int_hd, ipi_int_ff                        */
                           /*                                                           */
                           /*************************************************************/
                                if (traceback)
                                     interrupt_mask =  0x04;  // assert VCONFT:TB_INTEN
                                else if (history_dump)
                                     interrupt_mask =  0x40;  // assert VCONFT:HD_INTEN
                                else
                                     interrupt_mask =  0x08;  // assert VCONFT:FF_INTEN
                           /*************************************************************/
                           /* Update address registers                                  */
                           /*************************************************************/
                                WRITE_UINT32(vcop_regs->VIBSAR, (uint32)pointers->vibsar);
                                if ((ioctl & 0x30) == 0x10)
                                     WRITE_UINT32(vcop_regs->VPMFAR, (uint32)pointers->vpmfar);
                                if (traceback || history_dump)
                                     WRITE_UINT32(vcop_regs->VOBAR, (uint32)pointers->vobar);
                                if (session_params->ffData->initValue != 0x0000)
                                     WRITE_UINT32(vcop_regs->VPMISR, (uint32)session_params->ffData-
>vpmisr);
                                break;
                           } // if ((ioctl & 0x0C) == (0x04))
/*****************************************************************************/
/*                            Equalization                                   */
/*****************************************************************************/
                           else if (ioctl & 0x0C)
                           {
                                if (! session_params->updateVisrx)
                                     generate_s_params_and_mf ((vcopStruct*)vcop);
                                vcop_createConfig(configBTFD, (vcopStruct*)vcop);
                                if ((ioctl & 0x0C) == 0x08)
                                     WRITE_UINT32(vcop_regs->VAADAR, (uint32)pointers->vaadar);
                                if (ioctl & 0x01)
                                     traceback = TRUE;
                                if (ioctl & 0x02)
                                     history_dump = TRUE;
                           /*************************************************************/
                           /* Calculate the next interrupt and enable it                */
                           /* Only one interrupt is enabled according to this priority:  */
                           /* ipi_int_tb, ipi_int_hd, ipi_int_ff                        */
                           /*                                                           */
                           /*************************************************************/
```

```
                                if (traceback)
                                     interrupt_mask =  0x04;  // assert VCONFT:TB_INTEN
                                else if (history_dump)
                                     interrupt_mask =  0x40;  // assert VCONFT:HD_INTEN
                                else
                                     interrupt_mask =  0x08;  // assert VCONFT:FF_INTEN
                           /*************************************************************/
                           /* Update address registers                                  */
                           /*************************************************************/
                                WRITE_UINT32(vcop_regs->VIBSAR, (uint32)pointers->vibsar);
                                if ((ioctl & 0x30) == 0x10)
                                     WRITE_UINT32(vcop_regs->VPMFAR, (uint32)pointers->vpmfar);
                                if (traceback || history_dump)
                                     WRITE_UINT32(vcop_regs->VOBAR, (uint32)pointers->vobar);
                                if (session_params->ffData->initValue != 0x0000)
                                     WRITE_UINT32(vcop_regs->VPMISR, (uint32)session_params->ffData-
>vpmisr);
                                break;
                           } // if (ioctl & 0x0C)
/*****************************************************************************/
/*                            Traceback (+ History Dump)                     */
/*****************************************************************************/
                           else if (ioctl & 0x01)
                           {
                                interrupt_mask = 0x04;  // assert VCONFT:TB_INTEN
                                // Update VOBAR only if regular Traceback/History Dump
                                if (session_params->ffData->interimDumps == 0)
                                     WRITE_UINT32(vcop_regs->VOBAR, (uint32)pointers->vobar);
                                break;
                           }

/*****************************************************************************/
/*                            History Dump                                   */
/*****************************************************************************/
                           else if (ioctl & 0x02)
                           {
                                interrupt_mask = 0x40;  // assert VCONFT:HD_INTEN
                                WRITE_UINT32(vcop_regs->VOBAR, (uint32)pointers->vobar);
                                break;
                           }
/*****************************************************************************/
/*                            PM Fill                                        */
/*****************************************************************************/
                           else if (ioctl & 0x10)
                           {
                                interrupt_mask = 0x10;
                                WRITE_UINT32(vcop_regs->VPMFAR, (uint32)pointers->vpmfar);
                                break;
                           }
/*****************************************************************************/
/*                            PM Clear                                       */
/*****************************************************************************/
                           else if (ioctl & 0x20)
                           {
                                interrupt_mask = 0x20;
                                break;
                           }
/*****************************************************************************/
/*                            PM Dump                                        */
/*****************************************************************************/
```

```
                        else
                        {
                            interrupt_mask = 0x02;
                            WRITE_UINT32(vcop_regs->VAADAR, (uint32)pointers->vaadar);
                            break;
                        }
            } // switch (ioctl)
/*****************************************************************************/
/*              Program VCONFR and clear VSTR                                */
/*****************************************************************************/
            session_params->ffData->vconfr =
                   ((ioctl << 26) | (session_params->ffData->vconfr & 0x00FFFFFF));
            WRITE_UINT32(vcop_regs->VSTR, 0xFFFFFFFF);
            WRITE_UINT32(vcop_regs->VCONFR, session_params->ffData->vconfr | (interrupt_mask <<
2));
            return VCOP_SUCCESS;
}
```

4.2.2 Write Session Configuration to VCOP, vcop_CreateConfig

```
#include "vcopDriver.h"
void vcop_createConfig(configType type, vcopStruct* dev_ch)
{
            volatile VCOP *vcop_regs = (VCOP*)dev_ch->vcop_regs;
            volatile vcopFFStruct      *ffData;
            volatile vcopTBStruct      *tbData;
            volatile vcopInterimStruct *decodingVisrxData;
            volatile vcopSParamStruct  *equalizationVisrxData;
            volatile vcopLpuStruct     *lpuData;
            switch (type)
            {
                   // inConfig is ffData;
                   case configFF:
                          ffData = dev_ch->session->ffData;
                          WRITE_UINT32(vcop_regs->VPOLR,
                              (ffData->polynomial3)      |
                              (ffData->polynomial2 << 8) |
                              (ffData->polynomial1 << 16)|
                              (ffData->polynomial0 << 24));
                          if (ffData->puncturePeriod)
                              WRITE_UINT32(vcop_regs->VPPR,
                                   (ffData->puncturePeriod - 1) |
                                   (ffData->puncturePattern << (32 - ffData->puncturePeriod)));
                          else
                              WRITE_UINT32(vcop_regs->VPPR, 0x00000000);
                          ffData->vpmisr =
                              (((ffData->initState << 24) | 0xffff ) &
                               (ffData->initValue));
                          ffData->vconfr =
                              ((ffData->interimDumps << 11) |
                               ((ffData->constraintLength - 5) << 16) |
                               (ffData->min_max << 19) |
                               (((ffData->rate == 3) ? 1 :
                                 (ffData->rate == 4) ? 2 :
                                 (ffData->rate == 6) ? 3 : 0) << 22));
                          break;
                   // inConfig is tbStruct;
                   case configTB:
                          tbData = dev_ch->session->tbData;
```

```
                        WRITE_UINT32(vcop_regs->VIBBLR,
                            (tbData->inputSize));
                        WRITE_UINT32(vcop_regs->VCNT,
                            (tbData->tbState            |
                            (tbData->learnPeriod << 10) |
                            (tbData->tbMode      << 17) |
                            (tbData->tbStage     << 20)));
                        break;
                    // inConfig is interimStruct;
                    case configBTFD:
                        decodingVisrxData = (vcopInterimStruct*)dev_ch->session->visrxData;
                        WRITE_UINT32(vcop_regs->VISRA,
                            (decodingVisrxData->stage[3]         |
                            (decodingVisrxData->stage[2]  << 8)  |
                            (decodingVisrxData->stage[1]  << 16) |
                            (decodingVisrxData->stage[0]  << 24)));
                        WRITE_UINT32(vcop_regs->VISRB,
                            (decodingVisrxData->stage[7]         |
                            (decodingVisrxData->stage[6]  << 8)  |
                            (decodingVisrxData->stage[5]  << 16) |
                            (decodingVisrxData->stage[4]  << 24)));
                        WRITE_UINT32(vcop_regs->VISRC,
                            (decodingVisrxData->stage[11]        |
                            (decodingVisrxData->stage[10] << 8)  |
                            (decodingVisrxData->stage[9]  << 16) |
                            (decodingVisrxData->stage[8]  << 24)));
                        break;
                    // inConfig is interimStruct;
                    case configSParam:
                        equalizationVisrxData = (vcopSParamStruct*)dev_ch->session->visrxData;
                        WRITE_UINT32(vcop_regs->VISRA,
                            ((equalizationVisrxData->sParams[1] & 0x0000ffff) |
                            (equalizationVisrxData->sParams[0]  << 16)));
                         WRITE_UINT32(vcop_regs->VISRB,
                            ((equalizationVisrxData->sParams[3] & 0x0000ffff) |
                            (equalizationVisrxData->sParams[2]  << 16)));
                         WRITE_UINT32(vcop_regs->VISRC,
                            ((equalizationVisrxData->sParams[5] & 0x0000ffff) |
                            (equalizationVisrxData->sParams[4]  << 16)));
                        break;
                    // inConfig is lpuStruct;
                    case configLPU:
                        lpuData = dev_ch->session->lpuData;
                        WRITE_UINT32(vcop_regs->VPCR,
                            (lpuData->event     << 16) |
                            (lpuData->genInt    << 24) |
                            (lpuData->stopCond  << 25) |
                            (lpuData->startCond << 30));
                         break;
            }
            return;
}
       #include "vcopDriver.h"
// Function in Assembly from vcopDriverAsm.asm
extern void initialize_driver();
// Variables from vcopDriver.c
extern VCOP *vcop_place_holder;
```

4.2.3 Initialize the Driver, vcop_InitializeDriver

vcop_InitializeDriver configures the LIC and the PIC and also the driver view of the VCOP.

```
        vcopStatus vcop_initializeDriver(vcopStruct* dev_ch)
{
                initialize_driver();
                dev_ch->vcop_regs = vcop_place_holder;
        #ifdef DECODING_BTFD
                dev_ch->btfd_management.num_of_remaining_BTFD_tracebacks = 0;
                dev_ch->btfd_management.num_of_BTFD_TB_handled           = 0;
#endif
                DISABLE_VCOP_INTERRUPTS_LIC();
                return VCOP_SUCCESS;
}
```

The Assembly code for programming the LIC and PIC follows.

```
        ; @Cautions               The interrupt connecting the LIC and the PIC is hardcoded.
;                                 This version of the driver does not allow for changing this.
;
;*//***************************************************************************/

        section    .oskernel_text_run_time_critical
                include "common_macros.asm"
        BASE_EXEPTION_TABLE equ $00001000
        ; Defined in this file
                GLOBAL     _initialize_driver
        _initialize_driver:
                move.l #BASE_EXEPTION_TABLE,vba ; set VBA to some value
                bmclr #$00e0,sr.h
                nop
                nop
                ; un-mask all interrupts
                ;; Clear all pending interrupts
                write_w#$ffff,IPRA
                write_w#$ffff,IPRB
                write_l#$ffffffff,LICAISR
                write_l#$ffffffff,LICBISR
                ;; VCOPSD is No. 28 in LIC interrupt group A table (ipi_int_protocol)
                ;; VCOPGI is No. 27 in LIC interrupt group A table (ipi_int_abort)
                ;; programing the LIC to work in edge mode, and choose the IRQOUTA[0-3] to PIC
                ;; EM28   = 01 edge,second edge
                ;; IMAP28 = 00 route 28 through IRQOUTA[1]
                ;; EM27   = 01 edge,ignore second edge
                ;; IMAP27 = 00 route 27 through IRQOUTA[0]
                write_l #$00094000,LICAICR0
                ;; Enable interrupts 28 and 27 (group A)
                write_l #$18000000,LICAIER; enable irq 28 and 27 - VCOPSD, VCOPGI
                ;; LIC IRQOUTA[1] is goes to IRQ7 in PIC interrupt routine table
                ;; LIC IRQOUTA[0] is goes to IRQ6 in PIC interrupt routine table
                ;; The service routing address (offset from VBA) is #$980
                ;; Program the PIC
                write_w #$6700,ELIRB          ; LIC IRQOUTA0/IRQOUTA1: edge triggered, highest priority
                                              ; PED7 = 0 level
                                              ; {PIL70,PIL71,PIL72} = 110
                                              ; PED6 = 0 level
                                              ; {PIL60,PIL61,PIL62} = 111 = highest priority

                 write_w #$7000,ELIRF         ; Enable LICSEIRQ - LIC Second Edge IRQ (Groups A and B)
                                              ; PED23 = 0 level
                                              ; {PIL230,PIL231,PIL232) = 111 = highest priority
```

```
                rts
               endsec
```

4.3 Driver Header File

```
/*************************************************************************
       @Cautions              -   Keep in mind that stage 0 is the first stage when a stage is called
for.
                              -   The driver assumes that the user will perform an interim dump
                                  at the last stage of a decoding session that requires such dumps.
                                  Furthermore, the driver overrides automatic tracebacks that
                                  follow a decoding session with interim dumps.
                              -   The driver assumes that if VCONFR[ISRNUM] is different than 0,
                                  the interrupt handlers should treat the interrupt as
                                  part of a BTFD session. This is correct for traceback sessions as
well.
                              -   Illegal configurations may result in a system halt.
                              -   When an interrupt is triggered, the user's function is called.
                                  While this function is active, other hardware
                                  interrupts are masked. Therefore the user's callback function
                                  should be as short as possible.
*//************************************************************************/
       #ifndef VCOPDRIVER_H
#define VCOPDRIVER_H
/************************************************************************//**
@Description   Various user definitions for the configuration of the driver
*//************************************************************************/
       // The base address of the VCOP as the core sees it (See MSC8126 Spec for more details)
#define VCOP_BASE_ADDRESS        0x01FEF000
// The base address of the LIC/PIC as the core sees it (See MSC8126 Spec for more details)
#define QBUS_BASE_ADDRESS        0x00F09C00
       // Should be defined if the driver is to perform BTFD or GSM Equalization complementary tasks
#define USE_DSP_COMPLEMETARY_TASKS
#ifdef USE_DSP_COMPLEMETARY_TASKS
               // If there is prior knowledge as to the value of K (constraint length, VCONFR:CNST)
               // for ALL uses of the driver, VCOP_BTFD_PRE_KNOWLEDGE should be set to 2**K
               #define VCOP_BTFD_PRE_KNOWLEDGE 512
               // The original BTFD equation suggested in the standard is: -10(log(Sn)) < D.
               //This can be transformed in to Sn > D' where D' = 10**(-D/10). BTFD_THRESHOLD is D'
               #define BTFD_THRESHOLD           -0.5
               // When defined, the CRC checking function removes K-1 bits.
               // When not set, no bit removal occurs
               #define TAIL_BITS
#endif // USE_DSP_COMPLEMETARY_TASKS

       // All user callback functions are assumed to return with the type of
// USER_FUNCTION_RETURN_TYPE
typedef void* USER_FUNCTION_RETURN_TYPE;
// All user callback functions are assumed to receive arguments the type of
// USER_FUNCTION_VARS_TYPE
typedef void* USER_FUNCTION_VARS_TYPE;
       #include "vcop_datatypes.h"
       /************************************************************************//**
@Description   vcopStatus enumerates the statuses the driver may return to the user
*//************************************************************************/
```

```
        typedef enum vcopStatus
{
                // Notifies the user of a successful ending of a session
                VCOP_SUCCESS,
                // Notifies the user that the requested configuration failed
                ILLEGAL_CONFIG,
                // Notifies the user that the current session has been aborted
                SESSION_ABORTED,
                // Notifies the user that none of the Interim Dump/Traceback pairs are
                // candidates for a valid data
                BTFD_FAILURE,
                // Notifies the user that one of the Interim Dump/Traceback pairs contains valid data
                BTFD_SUCCESS,
                // Notifies the user that the driver could not be initialized
                INITIALIZATION_FAILURE,
} vcopStatus;
        /*****************************************************************************//**
@Description    minMax enumerates Min/Max PM calculations
                            This is used to set the ffStruct.min_max field to the desired
                            type of PM calculation
*//*****************************************************************************/
typedef enum minMax
{
                // Minimum PM calculations
                MIN = 1,
                // Maximum PM calculations
                MAX = 0
} minMax;
        /*****************************************************************************//**
@Description    vcopCRCType enumerates the various numbers of CRC bits
                                appended to the data. Default is CRC0
                                - gCRC24(D) = D24 + D23 + D6 + D5 + D + 1
                                - gCRC16(D) = D16 + D12 + D5 + 1
                                - gCRC12(D) = D12 + D11 + D3 + D2 + D + 1
                                - gCRC8(D)  = D8  + D7  + D4 + D3 + D + 1
*//*****************************************************************************/
typedef enum vcopCRCType
{
                // No CRC
                CRC0 = 0,
                // 8 bit CRC.  Polynomial is: D8  + D7  + D4 + D3 + D + 1
                CRC8 = 8,
                // 12 bit CRC. Polynomial is: D12 + D11 + D3 + D2 + D + 1
                CRC12 = 12,
                // 16 bit CRC. Polynomial is: D16 + D12 + D5 + 1
                CRC16 = 16,
                // 24 bit CRC. Polynomial is: D24 + D23 + D6 + D5 + D + 1
                CRC24 = 24
} vcopCRCType;
        /*****************************************************************************//**
@Description    vcopFFStruct is the user's view of the what the encoder
                                did before transmitting the data
*//*****************************************************************************/
typedef struct
{
                // Polynomial 0 as well as polynomial 3 (if vcopFFStruct.rate equals 6)
                uint8       polynomial0;
                // Polynomial 1 as well as polynomial 4 (if vcopFFStruct.rate equals 6)
                uint8       polynomial1;
                // Polynomial 2 (if vcopFFStruct.rate is larger than 2),
```

```
                // as well polynomial 5 (if vcopFFStruct.rate equals 6)
                uint8       polynomial2;
                // Polynomial 3 (if vcopFFStruct.rate equals 4)
                uint8       polynomial3;
                // The puncture pattern used when transmitting the encoded data. A '1' specifies an
                // unpunctured bit.
                uint32      puncturePattern;
                // The repetition factor of the puncture pattern. If there is no puncturing set to 0
                uint8       puncturePeriod;
                // The rate at which the transmitted bits were encoded. The legal values are {2, 3, 4, 6}
                uint8       rate;
                // The constraint length used when encoding the bits. The legal values are {5, 6, 7, 8,
9}
                uint8       constraintLength;
                // Defines whether the VCOP should look for the minimum or maximum Path Metrics
                minMax      min_max;
                // The number of interim dumps to be dumped during the Feed Forward session.
                // The legal values are 0-12. SEE GENERAL CAUTIONS OF vcopDriver.h!!!
                uint8       interimDumps;
                // The state to be initialized to the value of vcopFFStruct.initValue. This is used to
help
                // the VCOP reach the correct decision based on previous knowledge the user may have
                uint8       initState;
                // The value to which vcopFFStruct.initState should be initialized. If no initialization
is
                // required, set vcopFFStruct.initValue to 0x0000
                int16       initValue;
                // The CRC type used
                vcopCRCType crc_type;
                uint32 vpmisr;
                uint32 vconfr;
} vcopFFStruct;
        /*****************************************************************************//**
@Description   vcopTBStruct is the user's view of the Data part of the session
        *//*****************************************************************************/
typedef struct
{
                // The size of the input buffer (in bytes)
                uint16 inputSize;
                // The stage from which to perform the traceback. If automatic traceback is used, this
                // should reflect the last stage in the session
                uint16 tbStage;
                // The mode to use for the traceback. The legal values are 0-7. For details, see the VCOP
                // chapter in the MSC8126 Reference Manual.
                uint8  tbMode;
                // The period to be used for learning. This is used when "stitching" two blocks
together.
                // See the VCOP Spec for more information
                uint8  learnPeriod;
                // The state from which to start the Traceback (valid only for odd values of
                // vcopTBStruct.tbMode
                uint8  tbState;
} vcopTBStruct;
```

```
        /*************************************************************************//**
@Description   vcopInterimStruct is an array of (up to) 12 stages from which
                              to perform interim dumps
*//*************************************************************************/
typedef struct {
               // Each element of the array describes the stage at which an Interim Dump is required
(given
               // that vcopFFStruct.interimDumps enables it). The legal values are 0-255
               uint8  stage[12];
} vcopInterimStruct;
        /*************************************************************************//**
@Description   vcopInterimStage holds all relevant information needed to manage each interim
                              dump. This structure is for the internal use of the driver
        *//*************************************************************************/
        typedef struct
{
               // Notifies the driver whether a traceback has been performed from this stage
               bool    performed_traceback;
               // Notifies the driver whether the traceback result passed CRC
               bool    passed_CRC;
               // The value of Sn (see Eq. 1 in Section A.1.2 of 3GPP TS 25.212)
               fp32    Sn;
               // A pointer to the traceback resulting from this interim dump
               uint32  vobar;
} vcopInterimStage;
        /*************************************************************************//**
@Description   vcopBTFDManagement holds all relevant information required for running BTFD
        *//*************************************************************************/
        typedef struct
{
               vcopInterimStage stage_data[12];
               int8             num_of_remaining_BTFD_tracebacks;
               uint8            num_of_BTFD_TB_handled;
               uint16           PM_dump_size;
               uint32           vobar;
        } vcopBTFDManagement;
        /*************************************************************************//**
@Description   vcopLpuStruct holds the user's view of the LPU (Local Performance Monitor)
        *//*************************************************************************/
typedef struct
        {
               // When true, the LPU is reset before the new configuration is programmed
               bool   reset;
               // The starting trigger for the LPU counters (see VCOP Spec for more details)
               uint8  startCond;
               // The condition that stops the LPU counters (see VCOP Spec for more details)
               uint8  stopCond;
               // Useless in the MSC8126. Defines whether ipi_int_pmi should assert when the LPU stops
               // counting
               bool   genInt;
               // The event during which the LPU counters count (see the MSC8126 Reference Manual for
details)
               uint8  event;
} vcopLpuStruct;
        /*************************************************************************//**
@Description                   vcopAddressPointers holds pointers to addresses in the M2
                               These pointers are assumed to be updated whenever
                               vcop_Ioctl(void*, uint8, void*) is called
```

```
        *//*************************************************************************/
typedef struct
{
                // VCOP Output Address Register (see VCOP Spec for more details)
                uint32 vobar;
                // VCOP Input Buffer Start Address Register (see VCOP Spec for more details)
                uint32 vibsar;
                // VCOP PM Fill Address Register (see VCOP Spec for more details)
                uint32 vpmfar;
                // VCOP Algorithm Assist Dump Address Register (see VCOP Spec for more details)
                uint32 vaadar;
                uint32 equalization_pre_vibsar;
} vcopAddressPointers;
        /*************************************************************************//**
@Description   vcopSessionStruct holds the user's view of the entire session
*//*************************************************************************/
        typedef struct
{
                // A pointer to a struct of addresses
                volatile vcopAddressPointers *pointers;
                // Should be true if vcopSessionStruct.ffData has been updated since last session
                bool                         updateFF;
                // A pointer to the relevant vcopFFStruct
                volatile vcopFFStruct        *ffData;
                // Should be true if vcopSessionStruct.tbData has been updated since last session
                bool                         updateTb;
                // A pointer to the relevant vcopTBStruct
                volatile vcopTBStruct        *tbData;
                // Should be true if vcopSessionStruct.visrxData has been updated since last session. If
                // set to FALSE in Equalization sessions, the driver will calculate the S Parameters and
the
                // matched filter before activating the VCOP.
                bool                         updateVisrx;
                // A pointer to the relevant vcopInterimStruct
                volatile void                *visrxData;
                // Should be true if LPU should be used this session
                bool                         updateLpu;
                // A pointer to the relevant vcopLpuStruct
                volatile vcopLpuStruct       *lpuData;
        } vcopSessionStruct;
        /*************************************************************************//**
@Description   VCOP memory map (4Kb)
*//*************************************************************************/
typedef struct
{
                volatile uint32 VPOLR;
                volatile uint32 VPPR;
                volatile uint32 VCNT;
                volatile uint32 VOBAR;
                volatile uint32 VIBSAR;
                volatile uint32 VIBBLR;
                volatile uint32 VPMFAR;
                volatile uint32 VPMISR;
                volatile uint32 VISRA;
                volatile uint32 VISRB;
                volatile uint32 VISRC;
                volatile uint32 VAADAR;
                volatile uint32 reserved1[0x0004];
                volatile uint32 VCONFR;
                volatile uint32 VSTR;
```

```
                volatile uint8  reserved2[0x03AE];
                volatile uint32 VPCR;
                volatile uint32 VPCA;
                volatile uint32 VPCB;
                volatile uint8  reserved3[0x003D];
} VCOP;
        /*****************************************************************************//**
@Description   PIC and LIC as seen by the core's QBUS memory map
*//*****************************************************************************/
typedef struct
{
// PIC
                volatile uint16 ELIRA;/* Edge/Level-Triggered Interrupt Register A */
                volatile uint8  reserved20[6];
                volatile uint16 ELIRB;  /* Edge/Level-Triggered Interrupt Register B */
                volatile uint8  reserved21[6];
                volatile uint16 ELIRC;  /* Edge/Level-Triggered Interrupt Register C */
                volatile uint8  reserved22[6];
                volatile uint16 ELIRD;  /* Edge/Level-Triggered Interrupt Register D */
                volatile uint8  reserved23[6];
                volatile uint16 ELIRE;  /* Edge/Level-Triggered Interrupt Register E */
                volatile uint8  reserved24[6];
                volatile uint16 ELIRF;  /* Edge/Level-Triggered Interrupt Register F */
                volatile uint8  reserved25[6];
                volatile uint16 IPRA;   /* Interrupt Pending Register A */
                volatile uint8  reserved26[6];
                volatile uint16 IPRB;   /* Interrupt Pending Register B */
                volatile uint8  reserved27[14];
                volatile uint8  reserved28[4024];
        // LIC
                volatile uint32 LICAICR0;/*LIC Group A Interrupt Configuration Register 0 */
                volatile uint8  reserved29[4];
                volatile uint32LICAICR1;/*LIC Group A Interrupt Configuration Register 1 */
                volatile uint8  reserved30[4];
                volatile uint32LICAICR2;/*LIC Group A Interrupt Configuration Register 2 */
                volatile uint8  reserved31[4];
                volatile uint32LICAICR3;/*LIC Group A Interrupt Configuration Register 3 */
                volatile uint8  reserved32[4];
                volatile uint32LICAIER;/*LIC Group A Interrupt Enable Register*/
                volatile uint8  reserved33[4];
                volatile uint32LICAISR;/*LIC Group A Interrupt Status Register*/
                volatile uint8  reserved34[4];
                volatile uint32LICAIESR;/*LIC Group A Interrupt Error Status Register */
                volatile uint8  reserved35[12];
                volatile uint32LICBICR0;/*LIC Group B Interrupt Configuration Register 0 */
                volatile uint8  reserved36[4];
                volatile uint32LICBICR1;/*LIC Group B Interrupt Configuration Register 1 */
                volatile uint8  reserved37[4];
                volatile uint32LICBICR2;/*LIC Group B Interrupt Configuration Register 2 */
                volatile uint8  reserved38[4];
                volatile uint32LICBICR3;/*LIC Group B Interrupt Configuration Register 3 */
                volatile uint8  reserved39[4];
                volatile uint32LICBIER;/*LIC Group B Interrupt Enable Register16 */
                volatile uint8  reserved40[4];
                volatile uint32LICBISR;/*LIC Group B Interrupt Status Register16 */
                volatile uint8  reserved41[4];
                volatile uint32LICBIESR;/*LIC Group B Interrupt Error Status Register*/
                volatile uint8  reserved42[0x4f8C];
} msc8126_qbus_t;
```

```
        /*************************************************************************//**
@Description   vcopStruct is the driver's view of the VCOP and the session
*//*************************************************************************/
typedef struct
{
               // A pointer to the VCOP's base address. Set to VCOP_BASE_ADDRESS in
               // vcop_initializeDriver(vcopStruct*)
               volatile VCOP                *vcop_regs;
               // A pointer to the current session. This pointer is updated whenever
               // vcop_Ioctl(void*, uint8, void*) is called
               volatile vcopSessionStruct   *session;
               // A pointer to the function the user wants the driver to call when sessions end
               USER_FUNCTION_RETURN_TYPE   (*user_end_of_session)(USER_FUNCTION_VARS_TYPE,
USER_FUNCTION_VARS_TYPE);
               vcopBTFDManagement            btfd_management;
} vcopStruct;
        /*************************************************************************//**
@Description   vcopSessionStatus is a structure used to pass arguments to the SWI handlers
        This structure is for the internal use of the driver
*//*************************************************************************/
typedef struct
{
               vcopStruct        *dev_ch;
               vcopStatus         status;
} vcopSessionStatus;
/*************************************************************************//**
@Description   configType enumerates various configuration options
                                 Describes the function vcop_createConfig(configType, vcopStruct*)
                                 which "user friendly" configuration should be analyzed and
configured.
                                 While a user may call this function, there is no need for it since it
                                 is automatically called from vcop_Ioctl(void*, uint8, void*)
                                 depending on the values of:
                                 vcopSessionStruct.updateFF with configFF,
                                 vcopSessionStruct.updateTb with configTB,
                                 vcopSessionStruct.updateVisrx with configBTFD
                                 vcopSessionStruct.updateVisrx with configSParam
                                 vcopSessionStruct.updateLpu with configLPU
} vcopSessionStruct;
        *//*************************************************************************/
typedef enum configType
{
               // Analyze a vcopFFStruct structure
               configFF,
               // Analyze a vcopTBStruct structure
               configTB,
               // Analyze a vcopInterimStruct structure
               configBTFD,
               // Analyze a vcopSParamSrtuct structure
               configSParam,
               // Analyze a vcopLPUStruct structure
               configLPU
} configType;
```

```
        /*************************************************************************//**
@Function      vcop_initializeDriver
@Description   Initializes the Driver.
@Param         dev_ch          - A pointer to the driver's view of the VCOP and the session
@Return        The status of the initialization.
*//*************************************************************************/
vcopStatus vcop_initializeDriver(vcopStruct* dev_ch);
/*************************************************************************//**
@Function      vcop_createConfig
       @Description   Translates the user's view of the session to the VCOP's view.
       @Param         type            - The type of translation requested (use a value from the
                                                configType enumeration)
       @Param         dev_ch          - A pointer to the driver view of the VCOP and the session
       @Cautions             Each call programs one or more registers in the VCOP, so
                             unnecessary calls should be avoided because of timing.
       *//*************************************************************************/
       void       vcop_createConfig(configType type, vcopStruct *dev_ch);
       /*************************************************************************//**
@Function      vcop_Ioctl
       @Description   Programs the VCOP to perform a session.
       @Param         dev_ch               - A pointer to the driver view of the VCOP and the session
@Param         ioctl                        - The session to be run. The value of ioctl is equivalent
to the
                                              value of VCONFR:CMD. A list of values and their meaning
can be
                                              found in the VCOP chapter of the MSC1826 Reference
Manual.@Param         param                - A pointer to the current session to be run. This pointer
is
                                              copied to dev_ch->session.
       @Return        The status of the configuration.
       @Cautions             If the function does not return VCOP_SUCCESS, the user callback function
is
                             not called for this session.
       *//*************************************************************************/
       vcopStatus vcop_Ioctl(void *dev_ch, uint8 ioctl, void *param);
/*************************************************************************//**
@Function      generate_s_params_and_mf
       @Description          Calculates the S Parameters for equalization and performs the matched
                             filter on the incoming data.
       @Param         dev_ch          - A pointer to the driver view of the VCOP and the session.
       *//*************************************************************************/
void generate_s_params_and_mf (vcopStruct* dev_ch);
/*************************************************************************//**
@Function      ACK_IPI_INT_ABORT
@Description   Acknowledges an Abort interrupt.
*//*************************************************************************/
       void ACK_IPI_INT_ABORT();
/*************************************************************************//**
@Function      ACK_IPI_INT_PROTOCOL
       @Description   Acknowledges a Protocol interrupt.
       *//*************************************************************************/
void ACK_IPI_INT_PROTOCOL();
/*************************************************************************//**
       @Function      ACK_DOUBLE_EDGE_PROTOCOL
       @Description   Acknowledges a second edge Protocol interrupt.
       *//*************************************************************************/
void ACK_DOUBLE_EDGE_PROTOCOL();
/*************************************************************************//**
       @Function      ENABLE_VCOP_INTERRUPTS_LIC
       @Description   Enables VCOP interrupts via the LIC.
```

```
        *//*************************************************************************/
void ENABLE_VCOP_INTERRUPTS_LIC();
/*************************************************************************//**
        @Function       DISABLE_VCOP_INTERRUPTS_LIC
        @Description    Disables VCOP interrupts via the LIC.
        *//*************************************************************************/
void DISABLE_VCOP_INTERRUPTS_LIC();
/*************************************************************************//**
        @Function       session_done
        @Description            Resets the vcop handler (generally referred to as dev_ch) and
                                calls on the user's end-of-session function to notify the user
                                that the session has ended.
        @Param          result          - The result from the session.
        *//*************************************************************************/
void session_done(vcopStatus result);
/*************************************************************************//**
        @Function       HWI_vcop_abort_handler
        @Description    Handles an Abort interrupt.
        @Param          dev_ch          - A pointer to the driver's view of the VCOP and the session
        *//*************************************************************************/
void HWI_vcop_abort_handler (os_hwi_arg dev_ch);
        /*************************************************************************//**
        @Function       HWI_vcop_protocol_handler
        @Description    Handles a Protocol interrupt.
        @Param          dev_ch          - A pointer to the driver's view of the VCOP and the session
        *//*************************************************************************/
        void HWI_vcop_protocol_handler(os_hwi_arg dev_ch);
        /*************************************************************************//**
        @Function       HWI_vcop_protocol_second_edge_handler
        @Description    Handles a second edge Protocol interrupt.
        @Param          dev_ch          - A pointer to the driver's view of the VCOP and the session
        *//*************************************************************************/
void HWI_vcop_protocol_second_edge_handler(os_hwi_arg dev_ch);
        #define osGetCoreID() 0
#define READ_UINT8(data, arg)       data = (uint8)(arg)
#define READ_UINT16(data, arg)      data = (uint16)(arg)
#define READ_UINT32(data, arg)      data = (uint32)(arg)
#define GET_UINT8(arg)              (uint8)(arg)
#define GET_UINT16(arg)             (uint16)(arg)
#define GET_UINT32(arg)             (uint32)(arg)
#define WRITE_UINT8(arg, data)      arg = (uint8)(data)
#define WRITE_UINT16(arg, data)     arg = (uint16)(data)
#define WRITE_UINT32(arg, data)     arg = (uint32)(data)
        #endif //#ifndef VCOPDRIVER_H
```

4.4 Driver Source Code

```
        #include "vcopDriver.h"
        volatile vcopStruct *global_dev_ch;
                #define VCOP_PLACE_IN_MEM (void*)VCOP_BASE_ADDRESS
                VCOP *vcop_place_holder = (VCOP*)VCOP_PLACE_IN_MEM;
                #define QBUS_PLACE_IN_MEM (void*)QBUS_BASE_ADDRESS
                msc8126_qbus_t *qbus_place_holder = (msc8126_qbus_t*)QBUS_PLACE_IN_MEM;
        void ACK_IPI_INT_ABORT()
{
                WRITE_UINT32(qbus_place_holder->LICAISR, 0x08000000);
                WRITE_UINT16(qbus_place_holder->IPRA, 0x0040);
                return;
}
```

```
        void ACK_IPI_INT_PROTOCOL()
{
                WRITE_UINT32(qbus_place_holder->LICAISR, 0x10000000);
                WRITE_UINT16(qbus_place_holder->IPRA, 0x0080);
                return;
}
        void ACK_DOUBLE_EDGE_PROTOCOL()
{
                WRITE_UINT32(qbus_place_holder->LICAIESR, 0x10000000);
                WRITE_UINT16(qbus_place_holder->IPRB, 0x0080);
                return;
}
        void ENABLE_VCOP_INTERRUPTS_LIC()
{
                WRITE_UINT32(qbus_place_holder->LICAIER, (GET_UINT32((uint32)qbus_place_holder-
>LICAIER) | 0x18000000));
                return;
}
        void DISABLE_VCOP_INTERRUPTS_LIC()
{
                WRITE_UINT32(qbus_place_holder->LICAIER, (GET_UINT32((uint32)qbus_place_holder-
>LICAIER) & 0xE7FFFFFF));
                return;
}
        void session_done(vcopStatus result)
{
                global_dev_ch->btfd_management.num_of_remaining_BTFD_tracebacks = 0;
                global_dev_ch->btfd_management.num_of_BTFD_TB_handled           = 0;
                (*global_dev_ch->user_end_of_session)((USER_FUNCTION_VARS_TYPE)(global_dev_ch),
(USER_FUNCTION_VARS_TYPE)(result));
                DISABLE_VCOP_INTERRUPTS_LIC();
                return;
}
```

5 VCOP Code Examples

Examples in this section demonstrate how to set up a 3GPP and CDMA2000 session using the VCOP driver. All the functions called from within the code can be found in **Section 4.2**, *Driver Functions*, on page 22. **Section 5.4** shows a short interrupt handling session that also calls on functions described in **Section 4.2**.

5.1 VCOP Driver in GSM Equalization and GSM Decoding Session

```
        //@Description   An example of a user's application.
        //                      - First session - plain SOVA (16 stage recursive)
        //                      - Next session - GSM. 32 recursive SOVA leading to plain decoding

        #include "vcopDriver.h"
        #include "vcopCallBackFunctions.h"
        #include <stdio.h>
        #include "sova_test0_data.h"
        #include "gsm_test0_data.h"
        volatile bool send_next_session;
        /*************************************************************************/
        int main ()
        {
                vcopStruct        dev_ch;
                vcopStatus          vcop_status;
                vcopFFStruct        decoding0_ff,       decoding1_ff;
```

```
        vcopTBStruct        decoding0_tb,       decoding1_tb;
//      vcopInterimStruct   decoding0_btfd,     decoding1_btfd;
        vcopAddressPointers decoding0_pointers, decoding1_pointers;
        vcopSessionStruct   decoding0,          decoding1;
        vcopFFStruct        equalization0_ff,       equalization1_ff;
        vcopTBStruct        equalization0_tb,       equalization1_tb;
        vcopSParamStruct    equalization0_sparam,   equalization1_sparam;
        vcopAddressPointers equalization0_pointers, equalization1_pointers;
        vcopSessionStruct   equalization0,          equalization1;
        send_next_session = FALSE;
/*************************************************************************/
/*      VCOP Channel (dev_ch) Setup                                      */
/*************************************************************************/
        vcop_status = vcop_initializeDriver(&dev_ch);
        if (vcop_status != VCOP_SUCCESS)
        {
                printf("Couldn't initialize VCOP driver!!!\n");
                return 0;
        }
        dev_ch.user_end_of_session = cbSessionDone;
/*************************************************************************/
/*              Session Setup (GSM - SOVA part)                          */
/*************************************************************************/
        equalization1_ff.polynomial0      = 0;
        equalization1_ff.polynomial1      = 0;
        equalization1_ff.polynomial2      = 0;
        equalization1_ff.polynomial3      = 0;
        equalization1_ff.puncturePattern  = 0x000000;
        equalization1_ff.puncturePeriod   = 0;
        equalization1_ff.initState        = 0;
        equalization1_ff.initValue        = 0;
        equalization1_ff.rate             = 2;
        equalization1_ff.min_max          = MAX;
        equalization1_ff.constraintLength = 6;
        equalization1_ff.interimDumps     = 0;
        equalization1_tb.inputSize   = 228;
        equalization1_tb.tbStage     = 113;
        equalization1_tb.tbMode      = 6;
        equalization1_tb.learnPeriod = 0;
        equalization1_tb.tbState     = 0x00;
        equalization1_sparam.slot    = 1;
        equalization1_pointers.equalization_pre_vibsar = (uint32)&gsm_pre_vibsar0[0];
        equalization1_pointers.vibsar                  = (uint32)&gsm_vibsar0[0];
        equalization1_pointers.vaadar                  = (uint32)&gsm_vaadar0[0];
        equalization1_pointers.vobar                   = (uint32)&gsm_vobar0[0];
        equalization1.pointers           = &equalization1_pointers;
        equalization1.updateFF           = TRUE;
        equalization1.ffData             = &equalization1_ff;
        equalization1.updateTb           = TRUE;
        equalization1.tbData             = &equalization1_tb;
        equalization1.updateVisrx        = FALSE;
        equalization1.visrxData          = (void*)&equalization1_sparam;
        equalization1.updateLpu          = FALSE;
/*************************************************************************/
/*      Run the first part (SOVA) of the fourth session (GSM)            */
/*************************************************************************/
        vcop_status = vcop_Ioctl((void*)&dev_ch, 0x29, (void*)&equalization1);
        if (vcop_status != VCOP_SUCCESS)
        {
                printf("Couldn't run session SOVA part of session 4 (GSM)!!!\n");
```

```
                send_next_session = TRUE;
        }
/*************************************************************************/
/*      Session Setup (GSM - Decoding part)                              */
/*************************************************************************/
        decoding0_ff.polynomial0      = 0x09;
        decoding0_ff.polynomial1      = 0x0D;
        decoding0_ff.polynomial2      = 0;
        decoding0_ff.polynomial3      = 0;
        decoding0_ff.puncturePattern  = 0x000000;
        decoding0_ff.puncturePeriod   = 0;
        decoding0_ff.initState        = 0;
        decoding0_ff.initValue        = 0;
        decoding0_ff.rate             = 2;
        decoding0_ff.min_max          = MAX;
        decoding0_ff.constraintLength = 5;
        decoding0_ff.interimDumps     = 0;
        decoding0_ff.crc_type         = CRC0;
        decoding0_tb.inputSize   = 114;
        decoding0_tb.tbStage     = 113;
        decoding0_tb.tbMode      = 0;
        decoding0_tb.learnPeriod = 0;
        decoding0_tb.tbState     = 0x00;
        decoding0_pointers.vibsar   = (uint32)&gsm_vibsar0[0];
        decoding0_pointers.vobar    = (uint32)&gsm_vobar0[0];
        decoding0.pointers           = &decoding0_pointers;
        decoding0.updateFF           = TRUE;
        decoding0.ffData             = &decoding0_ff;
        decoding0.updateTb           = TRUE;
        decoding0.tbData             = &decoding0_tb;
        decoding0.updateVisrx        = FALSE;
        decoding0.updateLpu          = FALSE;
/*************************************************************************/
/*      Wait for Session (GSM - SOVA) to End                             */
/*************************************************************************/
        // The callback function will change send_next_session to TRUE
        while(send_next_session == FALSE);
        send_next_session = FALSE;
/*************************************************************************/
/*      Run the second part (Decoding) of the fourth session (GSM)       */
/*************************************************************************/
        vcop_status = vcop_Ioctl((void*)&dev_ch, 0x25, (void*)&decoding0);
        if (vcop_status != VCOP_SUCCESS)
        {
                printf("Couldn't run session Decoding part of session 4 (GSM)!!!\n");
                send_next_session = TRUE;
        }
/*************************************************************************/
/*      Wait for Fourth Session (GSM - Decoding) to End                  */
/*************************************************************************/
        // The callback function will change send_next_session to TRUE
        while(send_next_session == FALSE);
        send_next_session = FALSE;
/*************************************************************************/
/*      *******           End of program        *******                  */
/*************************************************************************/
        return 0;
}
```

5.2 Pre-Equalization Tasks

```
// @Description   Equlization preliminary activities.
//*************************************************************************/
#include "vcopDriver.h"
// Array of Training Sequences, one per slot
volatile const uint8 tsc[] = {

        0,0,1,0,0,1,0,1,1,1,0,0,0,0,1,0,0,0,1,0,0,1,0,1,1,1,
        0,0,1,0,1,1,0,1,1,1,0,1,1,1,1,0,0,0,1,0,1,1,0,1,1,1,
        0,1,0,0,0,0,1,1,1,0,1,1,1,0,1,0,0,1,0,0,0,0,1,1,1,0,
        0,1,0,0,0,1,1,1,1,0,1,1,0,1,0,0,0,1,0,0,0,1,1,1,1,0,
        0,0,0,1,1,0,1,0,1,1,1,0,0,1,0,0,0,0,0,1,1,0,1,0,1,1,
        0,1,0,0,1,1,1,1,0,1,0,1,1,0,0,0,0,0,0,1,0,0,1,1,1,0,1,0,
        1,0,1,0,0,1,1,1,1,1,1,0,1,1,0,0,0,1,0,1,0,0,1,1,1,1,1,1,
        1,1,1,0,1,1,1,1,1,0,0,0,1,0,0,1,0,0,1,0,1,1,1,0,1,1,1,1,0,0
};

volatile int8 vect[] = {
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,

        0,0,0,0,0,0,0,0,
        0,0
};
void generate_s_params_and_mf (vcopStruct* dev_ch)
{
        volatile vcopSessionStruct *session     = (vcopSessionStruct*)dev_ch->session;
        volatile vcopSParamStruct  *sParamData  = (vcopSParamStruct*)session->visrxData;
        int16             *sp          = (int16*)sParamData->sParams;
        int16             *data_buf    = (int16*)session->pointers->equalization_pre_vibsar;
        int16             *data_buf_mf = (int16*)session->pointers->vibsar;
        uint8 k = session->ffData->constraintLength;
        uint8 i,j;
        int16 *tsrx;     // Pointer to the TS in the data_buf
        int16 tsmf[26];  // TS after Matched Filter.
        int32 tsmf_tmp;  // Has to use 32 bits for the calculation,before devision
        int16 cir_[7];   // Effective Coeficiantes
        int32 sp_tmp;    // Has to use 32 bits for the calculation,before devision
        int16 tsrx_[121];// This is the actual data, without the TS, with some padding
        int32 data_buf_mf_tmp;
        // Pointer to the Training Sequence in the recieved data.
        tsrx = data_buf + 57;
        // Useing only the middle 16 bits of the TS. Flip their order and padd with
        //  zeros for convolution calculation.
        for (i=0; i<16; i++)
          vect[25+i] = 1 - 2 * (int8) tsc[(sParamData->slot * 26) + 20 - i];

        // Convolution
        for (i=0; i<26; i++)
        {
          tsmf_tmp = 0;
          for (j=0; j<26; j++)
            tsmf_tmp += tsrx[j] * vect[25 + i - j];
          tsmf[i] = tsmf_tmp / 16;
```

```
        } // for (i=0; i<26; i++)



        // Prepare the vector of effective coeficiantes for the S calculation.
        for (i=0; i<7; i++)
          if (i<k)
            cir_[i] = tsmf[20+i];
          else
            cir_[i] = 0;

        // SP Calculation
        for (i=0; i<6; i++)
        {
          if (i >= k)
            sp[i] = 0;
          else
          {
            sp_tmp = 0;
            for (j=0; j<k-1-i; j++)
              sp_tmp = sp_tmp + cir_[j]*cir_[j+i+1];
            sp[i] = sp_tmp / 16384;
          }
        }
        // Create the actual data vector
        // First remove the TS
        for (i=0; i<57; i++)
        {
          tsrx_[i] = data_buf[i];
          tsrx_[i+57] = data_buf[i+57+26];
        }

        // Than add padding for calculation purposes
        for (i=0; i<7; i++)
          tsrx_[i+114] = 0;

        // Perform the MAtched Filter
        for (i=0; i<114; i++)
        {

          data_buf_mf_tmp = 0;
          for (j=0; j<k; j++)
            data_buf_mf_tmp = data_buf_mf_tmp + cir_[j]*tsrx_[j+i];
//        data_buf_mf[i] = data_buf_mf_tmp / 16384;
          data_buf_mf[i] = data_buf_mf_tmp / 32768; // 2^15
          data_buf_mf[i] = ((data_buf_mf[i] & 0xFF00) >> 8) | ((data_buf_mf[i] & 0x00FF) << 8);

        }
        return;
} // generate_s_params_and_mf
```

5.3 Driver Functions for Handling SOVA

```
//@Description   Implementation of VCOP Driver internal functions for SOVA handling
*//*************************************************************************/
#include "vcopDriver.h"
#ifdef USE_DSP_COMPLEMETARY_TASKS
#include "vcopDriverComplementaryTasks.h"
```

```
// Soft Output Viterbi Algorithem

// ++++++++++++++++++++++++++++++

// tb_head : Pointer to the buffer of regular traceback

// tb_rec  : Pointer to the buffer of recursive traceback.

// delta   : Pointer to the buffer of Delta values

// tbndx   : Total number of stages [0..Num of stages-1]

// k       : Number of taps in filter (5,6,7)

// state   : Index number of state to start from [0..2^(k-1)-1]

// soft_output : Pointer to allocated memory buffer for results of SOVE, soft

//               output for the decoding stage, byte size signed values

void perform_SOVA_16 (vcopStruct* dev_ch)
{

#pragma opt_level = "O0"

        volatile vcopSessionStruct *session     = (vcopSessionStruct*)dev_ch->session;

        uint8             *tb_head     = (uint8*)session->pointers->vobar;

        uint8             *delta       = (uint8*)session->pointers->vaadar;

        int8              *soft_output = (int8*)session->pointers->vibsar;

        uint16             tbndx       = session->tbData->tbStage;

        uint8              k           = session->ffData->constraintLength;

        uint8              state;

        uint8* tb_rec;
        uint8 k_array[] = {0, 0, 0, 0, 8, 16, 32, 64};//Lookup table for num. of stages
        uint8* tb_ptr = tb_head; // Pointer to the relevant word in the TB buffer
        uint8 tb_word; // The efective TB word
        uint8 tb_next_word; // The next(previous in order) TB word from the buffer
        uint16 tb_internal_word; // TB word to be used within the recursion
        uint16 tb_internal_next_word; //
        uint16 rec_word; // The current Recursive word
        int16 current_stage; //

        static const int8 inf8 = 0x7f;

        if (session->tbData->tbMode & 0x1)

                state = session->tbData->tbState;

        else

        {

                uint32 vstr_tmp;
```

```
            READ_UINT32(vstr_tmp, dev_ch->vcop_regs->VSTR);

            state = (uint8)(vstr_tmp >> 24);

        }

    // Update the start of the traceback data according to size of dumped history buffer

        if (session->ffData->vconfr & 0x08000000)

        {

            tb_head += (k_array[k] * (tbndx + 1)) >> 3;

            tb_head += ((k_array[k] * (tbndx + 1)) & 0x003F) ? 1 : 0;

        }


        tb_rec = tb_head + (((tbndx >> 6) + ((tbndx & 0x003F) ? 1 : 0)) << 3);

        // Adjust according to bug of offset on recursive data start. Bug Id : MSIls13510

        tb_rec += 2*((k-2*(k-6)+(tbndx+1)) % 4);

        // Point to the end of the buffer because TB is done backwords.

        delta += tbndx * k_array[k];

        // Align the full TB to the starting point.

        // Prepare initial TB word and shift the next to be usefull.

        tb_ptr += 8 - ((tbndx & 0x003F) >> 3) - (tbndx & 0x07);

        tb_word = *tb_ptr++>>(8 - ((tbndx+1) & 0x07));

        tb_next_word = *tb_ptr++;

        tb_word += tb_next_word<<((tbndx+1) & 0x07);

        tb_next_word >>= (8 - ((tbndx+1) & 0x07));

        // Initialize the vector to infinity.
        for (current_stage=0; current_stage<=tbndx; current_stage++)
          soft_output[current_stage] = inf8;

        // This is the recursive loop
        for (current_stage=tbndx-(k-1); current_stage>0 ; current_stage--)
        {
          uint16 rec_stages;
          uint16 internal_ptr;
          // Prepare the recursive TB word
          rec_word = *tb_rec++ & 0x00FF;
          rec_word += ((*tb_rec++ & 0x00FF) << 8);

          // Prepare the current TB word
          tb_internal_word = *tb_ptr;
```

```
    tb_internal_word <<= ((current_stage+k) & 0x07);

    tb_internal_word += tb_next_word;

    tb_internal_word <<= 8-(k-1);

    tb_internal_word += (tb_word >> (k-1));

    tb_internal_next_word = *(tb_ptr+1);

    tb_internal_next_word <<= (16 - (k-1) + ((current_stage+k) & 0x07));

    tb_internal_word += tb_internal_next_word;

if (current_stage == 0)
  rec_stages = 1;
else if (current_stage < 16)
     rec_stages = current_stage;
   else
     rec_stages = 16;
   for (internal_ptr=0; internal_ptr<rec_stages; internal_ptr++)
   {
     if (((tb_internal_word & 0x01) != (rec_word & 0x01)) &&

         (soft_output[current_stage-internal_ptr] > (int8)(delta[state] >> 1)))
       soft_output[current_stage-internal_ptr] = (int8)(delta[state] >> 1);
     tb_internal_word>>=1;
     rec_word>>=1;
   } // internal_ptr

   delta -= k_array[k];
   state = (state>>1) + (k_array[k-1])*((tb_word>>(k-1)) & 0x01);
   if (!((current_stage+k) & 0x07)) // Word is empty, get the next word from mem.
     tb_next_word = *tb_ptr++;
   tb_word >>= 1;
   tb_word += (tb_next_word & 0x01)<<7;
   tb_next_word>>=1;
 } // The recursive loop

 // Quantize the vector - devide by 2 and add sign
 //for (current_stage=0; current_stage<=tbndx; current_stage++)
 //  soft_output[current_stage] /= 2;
 // Sign the vector according to the TB bits.
 tb_ptr = tb_head + 8 - ((tbndx & 0x003F) >> 3) - (tbndx & 0x07);
 tb_word = *tb_ptr++>>(8 - ((tbndx+1) & 0x07));
 tb_next_word = *tb_ptr++;
 tb_word += tb_next_word<<((tbndx+1) & 0x07);
 tb_next_word >>= (8 - ((tbndx+1) & 0x07));
 for (current_stage=tbndx+1; current_stage>0 ; current_stage--)
 {
   if (tb_word & 0x01)
        soft_output[current_stage-1] = -soft_output[current_stage-1];
   if (!((current_stage) & 0x07)) // Word is empty, get the next word from mem.
     tb_next_word = *tb_ptr++;
   tb_word >>= 1;
   tb_word += (tb_next_word & 0x01)<<7;
   tb_next_word>>=1;
 } // Sign the vector
 return;
```

```
} // perform_SOVA_16

// tb_head : Pointer to the buffer of regular traceback
// tb_rec  : Pointer to the buffer of recursive traceback.
// delta   : Pointer to the buffer of Delta values
// tbndx   : Total number of stages [0..Num of stages-1]
// k       : Number of taps in filter (5,6,7)
// state   : Index number of state to start from [0..2^(k-1)-1]
// soft_output : Pointer to allocated memory buffer for results of SOVE, soft
//               output for the decoding stage, byte size signed values
void perform_SOVA_32 (vcopStruct* dev_ch)
{
#pragma opt_level = "O0"
       volatile vcopSessionStruct *session     = (vcopSessionStruct*)dev_ch->session;
       uint8             *tb_head     = (uint8*)session->pointers->vobar;
       uint8             *delta       = (uint8*)session->pointers->vaadar;
       int8              *soft_output = (int8*)session->pointers->vibsar;
       uint16             tbndx       = session->tbData->tbStage;
       uint8              k           = session->ffData->constraintLength;
       uint8              state;

       uint8* tb_rec;
       uint8 k_array[] = {0, 0, 0, 0, 8, 16, 32, 64};//Lookup table for num. of stages
       uint8* tb_ptr = tb_head; // Pointer to the relevant word in the TB buffer
       uint8 tb_word; // The efective TB word
       uint8 tb_next_word; // The next(previous in order) TB word from the buffer
       uint32 tb_internal_word; // TB word to be used within the recursion
       uint32 tb_internal_next_word; //
       uint32 rec_word; // The current Recursive word
       int16 current_stage; //
       static const int8 inf8 = 0x7f;

       if (session->tbData->tbMode & 0x1)

              state = session->tbData->tbState;
       else
       {
              uint32 vstr_tmp;
              READ_UINT32(vstr_tmp, dev_ch->vcop_regs->VSTR);
              state = (uint8)(vstr_tmp >> 24);
       }

    // Update the start of the traceback data according to size of dumped history buffer
       if (session->ffData->vconfr & 0x08000000)
       {
              tb_head += (k_array[k] * (tbndx + 1)) >> 3;
              tb_head += ((k_array[k] * (tbndx + 1)) & 0x003F) ? 1 : 0;
       }
       tb_rec = tb_head + (((tbndx >> 6) + ((tbndx & 0x003F) ? 1 : 0)) << 3);

       // Adjust according to bug of offset on recursive data start. Bug Id : MSIls13510
       tb_rec += 4*((k+tbndx) % 2);

       // Point to the end of the buffer because TB is done backwords.
       delta += tbndx * k_array[k];

       // Align the full TB to the starting point.
       // Prepare initial TB word and shift the next to be usefull.
       tb_ptr += 8 - ((tbndx & 0x003F) >> 3) - (tbndx & 0x07);
       tb_word = *tb_ptr++>>(8 - ((tbndx+1) & 0x07));
```

```
  tb_next_word = *tb_ptr++;
  tb_word += tb_next_word<<((tbndx+1) & 0x07);
  tb_next_word >>= (8 - ((tbndx+1) & 0x07));

  // Initialize the vector to infinity.
  for (current_stage=0; current_stage<=tbndx; current_stage++)
    soft_output[current_stage] = inf8;
  // This is the recursive loop
  for (current_stage=tbndx-(k-1); current_stage>=0 ; current_stage--)
  {
    uint16 rec_stages;
    uint16 internal_ptr;



    // Prepare the recursive TB word
    rec_word = *tb_rec++ & 0x00FF;
    rec_word += ((*tb_rec++ & 0x00FF) << 8);
    rec_word += ((*tb_rec++ & 0x00FF) << 16);
    rec_word += ((*tb_rec++ & 0x00FF) << 24);

    // Prepare the current TB word
    tb_internal_word = *tb_ptr;
    tb_internal_word += *(tb_ptr+1) << 8;
    tb_internal_word += *(tb_ptr+2) << 16;
    tb_internal_word <<= ((current_stage+k) & 0x07);
    tb_internal_word += tb_next_word;
    tb_internal_word <<= 8-(k-1);
    tb_internal_word += (tb_word >> (k-1));
    tb_internal_next_word = *(tb_ptr+3);
    tb_internal_next_word <<= (32 - (k-1) + ((current_stage+k) & 0x07));
    tb_internal_word += tb_internal_next_word & 0xff000000;

if (current_stage == 0)

   rec_stages = 1;
else if (current_stage < 32)
      rec_stages = current_stage;
    else
      rec_stages = 32;
    for (internal_ptr=0; internal_ptr<rec_stages; internal_ptr++)
    {
      if (((tb_internal_word & 0x01) != (rec_word & 0x01)) &&
          (soft_output[current_stage-internal_ptr] > (int8)(delta[state] >> 1)))
        soft_output[current_stage-internal_ptr] = (int8)(delta[state] >> 1);
      tb_internal_word>>=1;
      rec_word>>=1;
    } // internal_ptr
    delta -= k_array[k];
    state = (state>>1) + (k_array[k-1])*((tb_word>>(k-1)) & 0x01);
    if (!((current_stage+k) & 0x07)) // Word is empty, get the next word from mem.
      tb_next_word = *tb_ptr++;
    tb_word >>= 1;
    tb_word += (tb_next_word & 0x01)<<7;
    tb_next_word>>=1;
  } // The recursive loop

  // Quantize the vector - devide by 2 and add sign
  //for (current_stage=0; current_stage<=tbndx; current_stage++)
  //  soft_output[current_stage] /= 2;
```

```
            // Sign the vector according to the TB bits.
            tb_ptr = tb_head + 8 - ((tbndx & 0x003F) >> 3) - (tbndx & 0x07);
            tb_word = *tb_ptr++>>(8 - ((tbndx+1) & 0x07));
            tb_next_word = *tb_ptr++;
            tb_word += tb_next_word<<((tbndx+1) & 0x07);
            tb_next_word >>= (8 - ((tbndx+1) & 0x07));
            for (current_stage=tbndx+1; current_stage>0 ; current_stage--)
            {
              if (tb_word & 0x01)
                   soft_output[current_stage-1] = -soft_output[current_stage-1];
              if (!((current_stage) & 0x07)) // Word is empty, get the next word from mem.
                tb_next_word = *tb_ptr++;
              tb_word >>= 1;
              tb_word += (tb_next_word & 0x01)<<7;
              tb_next_word>>=1;
            } // Sign the vector
            return;
      } // perform_SOVA_32
      #endif // USE_DSP_COMPLEMETARY_TASKS
```

5.4 Interrupt Handling

```
      // VCOP driver include file
#include "vcopDriver.h"
      /*****************************************************************************/
/*                          ipi_int_abort Handler                                */
/*****************************************************************************/
void HWI_vcop_abort_handler (os_hwi_arg dev_ch)
{
            // Acknowledge the Interrupt
            ACK_IPI_INT_ABORT();
            // Call the VCOP drivers' end of session function
            session_done(SESSION_ABORTED);
            return;
}
      /*****************************************************************************/
/*                          ipi_int_protocol Handler                             */
/*****************************************************************************/
      void HWI_vcop_protocol_handler(os_hwi_arg dev_ch)
      {
            // Acknowledge the Interrupt
            ACK_IPI_INT_PROTOCOL();
            // Call the VCOP drivers' end of session function
            session_done(VCOP_SUCCESS);
            return;
}
```

5.5 VCOP Interrupt Handlers for Equalization

```
      // @Description   Handling of hardware interrupts
      //      The interrupt handlers call on functions to
      //      perform the actual handling.
      #include "vcopDriver.h"
      #ifdef USE_DSP_COMPLEMETARY_TASKS
            #include "vcopDriverComplementaryTasks.h"
      #endif
      extern volatile vcopStruct *global_dev_ch;
      volatile bool continue_int_handling;
      /*****************************************************************************/
```

```
        /*                          ipi_int_abort Handler                                */
        /*****************************************************************************/
        void HWI_vcop_abort_handler (os_hwi_arg dev_ch)
        {

                ACK_IPI_INT_ABORT();
                session_done(SESSION_ABORTED);
                return;
        }

        /*****************************************************************************/
        /*                          ipi_int_protocol Handler                             */
        /*****************************************************************************/
        void HWI_vcop_protocol_handler(os_hwi_arg dev_ch)
        {
        #ifdef USE_DSP_COMPLEMETARY_TASKS
                /*************************************************************/
                /* Handle an interrupt with Interim Dumps specially          */
                /*************************************************************/
                if (global_dev_ch->session->ffData->interimDumps)
                {
                        // If ipi_int_ff, set the tracebacks in to motion
                        continue_int_handling = FALSE;
                        perform_BTFD_traceback(global_dev_ch, 0);
                        while (global_dev_ch->btfd_management.num_of_remaining_BTFD_tracebacks > 0)
                        {
                                while (continue_int_handling == FALSE);
                                continue_int_handling = FALSE;
                                perform_CRC_on_stage(global_dev_ch);
                        }
                        // analyse the last stage
                        perform_CRC_on_stage(global_dev_ch);
                        analyze_all_BTFD_TB(global_dev_ch);
                }

                /*************************************************************/

                /* Handle an interrupt with SOVA specially                   */

                /*************************************************************/
                else
                if ((global_dev_ch->session->tbData->tbMode & 0x04) && (global_dev_ch->session->ffData-
>vconfr & 0x04000000))
                {
                        if (global_dev_ch->session->tbData->tbMode & 0x02)
                                perform_SOVA_32((vcopStruct*)global_dev_ch);
                        else
                                perform_SOVA_16((vcopStruct*)global_dev_ch);
                        session_done(VCOP_SUCCESS);
                }
                /*************************************************************/
                /* Handle a regular interrupt                                */
                /*************************************************************/
                else
        #endif // USE_DSP_COMPLEMETARY_TASKS
                        session_done(VCOP_SUCCESS);
                ACK_IPI_INT_PROTOCOL();
                return;
        }

        /*****************************************************************************/
```

```
/*                ipi_int_protocol Second Edge Handler                          */
/*******************************************************************************/
void HWI_vcop_protocol_second_edge_handler(os_hwi_arg dev_ch)
{
        continue_int_handling = TRUE;
        ACK_DOUBLE_EDGE_PROTOCOL();
        return;
}
```

6 Local Profiling Unit (LPU)

The local profiling unit of the VCOP contains three registers, as follows:

- VPCR is the control register for the profiling monitor unit of the VCOP. The register should be programmed while the VCOP is in the IDLE mode, prior to issuing any command to the VCONFR.
- Viterbi Coprocessor Performance Monitor Counter A (VPCA). Counts the total number of cycles for this session of performance monitoring. This register can only be read when the performance monitoring session is complete. The value is read from inside the VCOP, so it is updated according to the internal clock. The data is stable and valid to be read only after the interrupt is issued.
- Viterbi Coprocessor Performance Monitor Counter B (VPCB). Counts the total number of cycles for the monitored event. For example, it counts the total number of cycles the VCOP spends waiting for ipm_req_ack. When the number of cycles of the monitored event is divided by the total number of cycles from the VPCA, the result is a percentage of time used for the monitored event. This register can be read only when the performance monitoring session completes. The value is read from inside the VCOP, so it is updated according to the internal clock. The data is stable and valid to be read only after the interrupt is issued.

7 References

[1] Viterbi, Andrew J. “Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm.” *Information Theory, IEEE Transactions on Information Theory,* Volume: 13, Issue: 2, April. 1967, pp. 260–269.

[2] Sklar Bernard. *Digital Communications, Fundamentals and Applications*. 2nd Edition, 2001, pp. 381–430.

[3] “3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Multiplexing and channel coding (FDD).” *3GPP TS 25.212.* V5.5.0, May, 2003.

[4] “3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Services provided by the physical layer.” *3GPP TS 25.302*. V5.1.0, May, 2002.

[5] Walid K. M. Ahmed. “Maximum-Likelihood Block-Size Detection for MPSK Signaling”. *IEEE Transactions on Vehicular Technology,* Volume 51, Issue 3, May 2002, pp. 260–269.

[6] MSC8126 *Reference Manual* (MSC8126RM), Freescale Semiconductor, Inc. Available at the web site listed on the back cover of this application note.

[7] *Initializing the MSC8101 Communications Processor Module and Using Pin_mux8101* (AN1854). This Freescale application note describes how to use Pin_mux8101 to initialize the MSC8101 CPM I/O. Both this application note and the software that accompanies it are available at the web site listed on the back cover of this application note.

[8] Hagenauer, J. and Hoeher, P., “A Viterbi Algorithm with Soft-Decision Outputs and its Applications”, *Global Telecommunications Conference*, 1989, and Exhibition. “Communications Technology for the 1990s and Beyond,” *IEEE GLOBECOM’89*, vol.3, Nov. 1989, pp. 1680–1686.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations not listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GMBH
Technical Information Center
Schatzbogen 7
81829 München, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
+800 2666 8080

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Order No.: AN2943
Rev. 0
5/2005