Application Note, V 1.1, Sept. 2001

AP32018

TriCore

Viterbi Decoding for V.32 standard

Microcontrollers

Never stop thinking.

Revision History:	**Sept. 2001**	**V 1.1**
Previous Version:	Feb. 1999	V 1.0

Page	Subjects (major changes since last revision)
all	Changed layout to Infineon Corporate Design

1 Abstract

In most wireless communications systems, convolutional coding is the preferred method of error-correction coding to overcome transmission distortions.

Practical applications of convolutional encoding became possible when **Viterbi** proposed a maximum-likelihood method for decoding convolutional codes in 1967.

This application note deals with encoding and decoding algorithms as required for the **V.32 standar**d. The basic encoding algorithm is known as a convolutional encoding scheme and the decoding algorithm scheme is based on the **Viterbi algorith**m.

In a first part the theoretical context is outlined:

- Encoder / decoder schematic block diagram
- Differential coding and decoding
- Convolutional encoding and decoding

Then the present implementation of both encoding and decoding algorithms is introduced in a second part.

Appendix contains both the C code implementation of the V.32 encoding and decoding algorithms and the benchmark for the complete decoding algorithm.

2 Theoretical Context

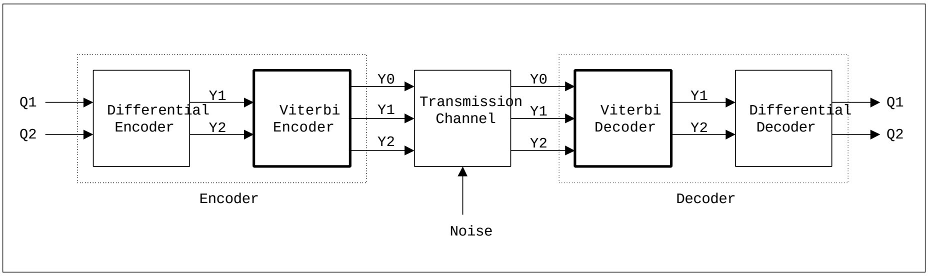


Figure 1 Encoder / Decoder Schematic Block Diagram

The V.32 encoder is divided into 2 functional blocks:

- Differential encoder
- Convolutional encoder also called Viterbi encoder

Decoding must be done by performing each decoder function in the reverse order in which it was encoded. Therefore, we have 2 functional blocks for decoding:

- Differential decoder
- Convolutional decoder also called Viterbi decoder

2.1 Differential Coding and Decoding Overview

Algorithms used both for differential encoding and decoding can be described by the following equations:

- Differential Encoder: $Y1_n = Q1_n \wedge Y1_{n-1}$
  $Y2_n = (Q1_n \bullet Y1_{n-1}) \wedge Y2_{n-1} \wedge Q2_n$
- Differential Decoder: $Q1_n = Y1_n \wedge Y1_{n-1}$
  $Q2_n = (Q1_n \bullet Y1_{n-1}) \wedge Y2_{n-1} \wedge Y2_n$

Note: (^) means EXCLUSIVE OR function.
(•) means AND function.

Refer to the parts called ”Differential Encoder Implementation” (chapter 4.1) and ”Differential Decoding Implementation” (chapter 7) for more details.

2.2 Convolutional Encoding and Decoding Overview

If convolutional encoding is easy to implement, however convolutional decoding is more complex.

2.2.1 Viterbi Encoding

The encoding method is referred to as convolutional coding. The outputs [Y0 Y1 Y2] are generated by convolving a signal [Y1 Y2] with itself, which adds a level of dependence on past values.

Convolutional encoder error-correction capabilities result from **outputs that depend on a sequence of past symbol value**s.

A simplified diagram of the Viterbi convolutional encoder is shown on the following figure:

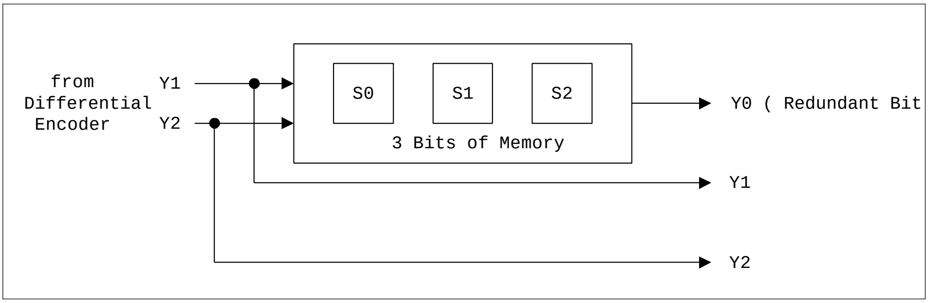


Figure 2 Viterbi Encoder

Definitions

- The 3 bits [S0 S1 S2] are called delay states and represent the state of the encoder.
- The 3 bits [Y0 Y1 Y2] are known as transitions and represent the encoded symbols that are output from the encoder. These 3-bit encoded symbols are transmitted, disturbed by the channel noise and then received by the decoder.
- Since the convolutional encoder is made of M = 3 bits of memory for V.32 standard, the constraint length K of the code is K = M + 1 = 4.
- The rate of this convolutional encoder is 2/3: 2 input bits [Y1 Y2] are encoded in a 3 bit transition [Y0 Y1 Y2].

Note: Refer to the part following the trellis diagram called "Vocabulary conventions" for more information about these concepts.

Constraint condition

Given a particular set of delay states [S0 S1 S2], not all transitions are possible in that time interval. For instance, given a delay state [0 0 1] for the encoder, only 4 transitions [0 0 0], [0 1 0], [1 0 0], and [1 1 0] are allowed in next time interval (see the trellis diagram, Figure 3).

This leads to the concept of trellis structure. Since the encoder is essentially a finite-state machine, a finite-state diagram may be used to represent it.

The following trellis diagram concisely illustrates possible transformations from one delay state to another, along with their corresponding transition:

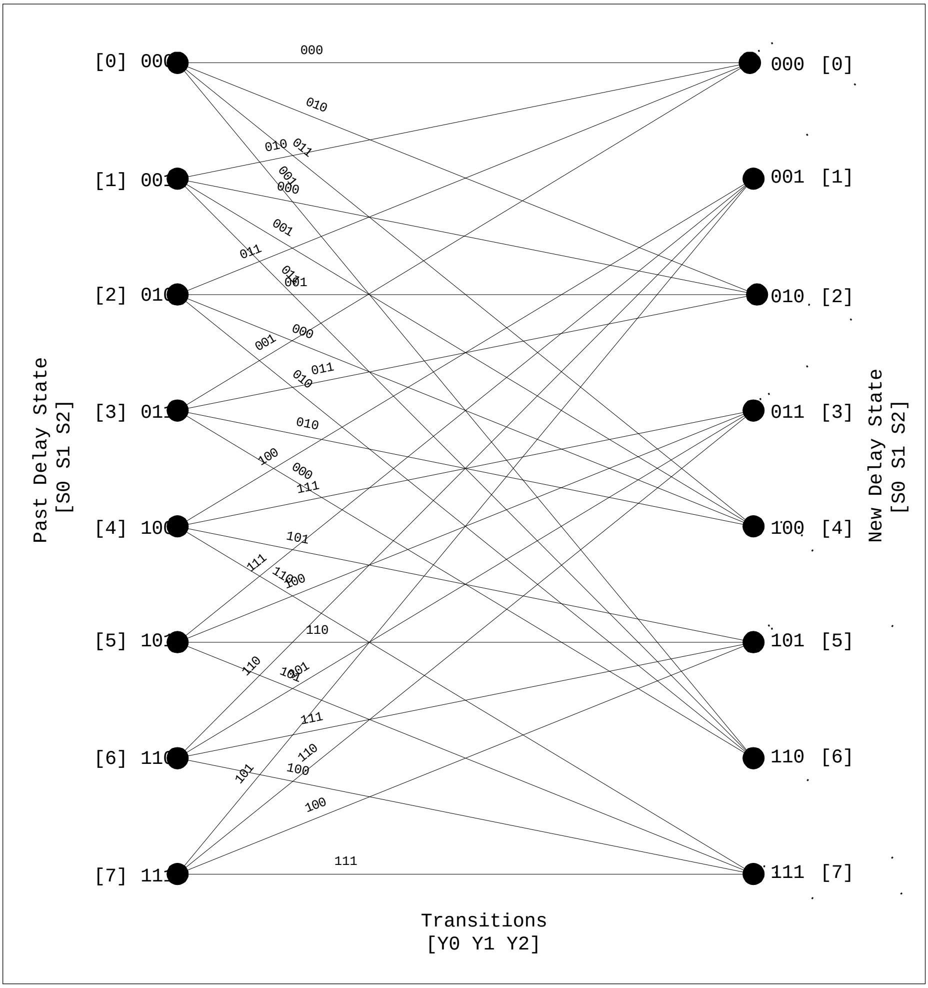


Figure 3 V.32 Modem Trellis Diagram

Note: Each column of delay states on the trellis diagram indicates one symbol interval.

2.2.2 Vocabulary conventions

Symbol

A **symbol** is a 2-bit value [Q1 Q2] and denotes a data that has to be sent. To protect it against channel distorsions, this data has to be encoded in a 3-bit **transition** [Y0Y1Y2] before it is sent. The aim of the decoder is then to decode the received transition so as to retrieve the transmitted **symbol**.

Delay State

the 3 bits [S0 S1 S2] in the encoder memory are called **delay states** since they represent the state of the 3 delays composing the convolutional encoder. For the 2/3 rate Viterbi encoder, there are 8 possible delay states, numbered from 0 to 7.
[see figure 2]

Branch

A **branch** is a link between 2 delay states. We have 32 different branches at each time period, divided in 8 groups of 4 branches: each group starts from one of the 8 **delay states**.
[see figure 3]

*Note: A branch is just a physical support to draw a **transition** in the trellis diagram. Its aim is especially to identify the transition it represents.*

Transition

the 3 bits [Y0 Y1 Y2] are known as **transitions** and represent the encoded symbols that are output from the encoder. A **transition** represents a sample transmitted by the encoder, disturbed in the channel, and received by the decoder.
There are 8 possible transitions numbered from 0 to 7.
[see figure 3]

Note: There are only 8 possible transitions for 32 branches: a transition identifies 4 differentbranches in the trellis diagram.

Note: Knowing the transition and the past delay state it comes from, the branch identified is unique.

Path

A **path** consists in a succession of linked **branches**, one branch for each time period.
2 successive branches of a path are connected by a **delay state**.
On the figure 4, 8 different paths have been drawn.
[see figure 4]

2.2.3 Convolutional Decoding Process

Convolutionally encoded data is decoded through knowledge of the possible state transitions (represented by the trellis diagram), created from the dependence of the current transition on past transitions. The decoding scheme makes use of past history and reliability information to decode incoming transitions.

2.2.4 Viterbi algorithm

This algorithm is based on a **maximum-likelihood decoding technique and was devised by A. J. Viterbi in 1967**: the decoder uses the trellis structure and continually calculates the distance between received and valid transitions.The purpose is to identify the transition sequence with the highest probability of matching the transmitted sequence based on the received sequence.

The encoder may attain only one delay state at any given time, but the decoder keeps track of all the possible delay states until it decides which one to select. **This is the essence of this algorithm in which the actual decision is delayed until more information is available**.

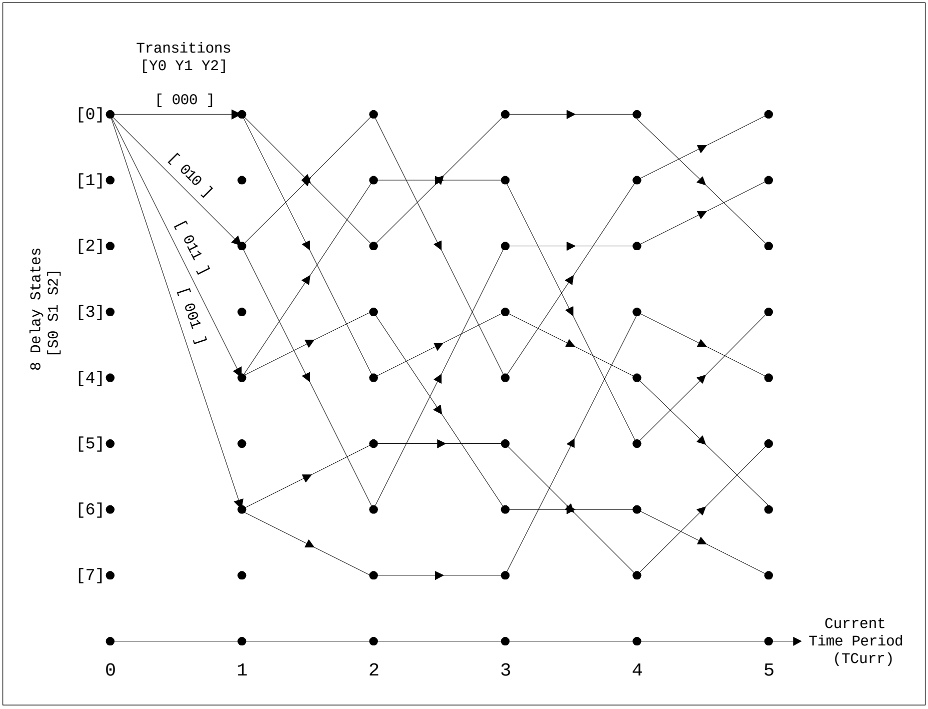


Figure 4 Dynamic Programming

Figure 4 shows an expanded trellis diagram over several transition time intervals with the x axis representing time and the y axis representing the eight possible delay states of the encoder.

In relation with the trellis diagram on figure 3, note that there are only 8 surviving branches for each time period instead of 32. In fact, considering the trellis diagram shows that 4 branches lead to every new delay state.

As proposed by Viterbi, the decision is performed at each time increment which is the branch belonging to the most likely path leading to the considered delay state. Only this branch is stored whereas the 3 others are discarded. Thus, the number of branches to store at each time period is reduced to 8, one for each delay state and **only 8 paths are stored in memory** after several time period as shown on figure 4.

Ideally, the maximum-likelihood method looks at the entire sequence of input transitions before making any decision about the output transitions. Clearly, this approach is not feasible for real-time applications due to two factors:

- Prohibitive memory requirements, even for relatively small blocks of data
- Inherent time delay before the decoder selects an output

A more practical approach is to consider only a finite length of input transitions before making a decision about the output. This length will be called **LENGTH_TB** in this application note.

Note: This length LENGTH_TB must be great enough to avoid deciding on a wrong path. This parameter value is determined by the constraint length K of the code (in our case, K = 4) and for near-optimum decoding should be chosen four or five times the constraint length. Since four times the constraint length in this case is 16 (4 x K), this makes modulo addressing easier than using 20 (5 x K) because 16 is a power of two.
So ***LENTH_TB = 16 is suitable for our applicatio****n.*

3 Implementation Overview

3.1 Data Flow Diagrams

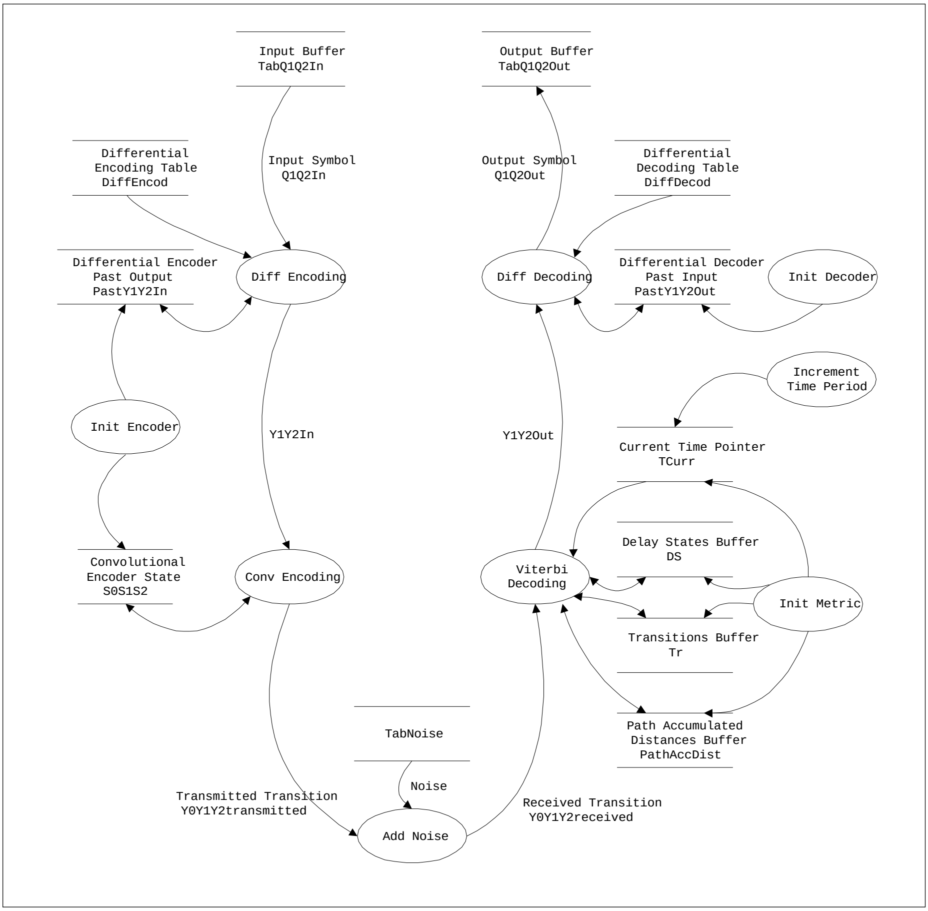

Input Buffer TabQ1Q2In
Output Buffer TabQ1Q2Out
Differential Encoding Table DiffEncod
Input Symbol Q1Q2In
Output Symbol Q1Q2Out
Differential Decoding Table DiffDecod
Differential Encoder Past Output PastY1Y2In
Diff Encoding
Diff Decoding
Differential Decoder Past Input PastY1Y2Out
Init Decoder
Increment Time Period
Init Encoder
Y1Y2In
Y1Y2Out
Current Time Pointer TCurr
Delay States Buffer DS
Convolutional Encoder State S0S1S2
Conv Encoding
Viterbi Decoding
Init Metric
Transitions Buffer Tr
TabNoise
Path Accumulated Distances Buffer PathAccDist
Noise
Transmitted Transition Y0Y1Y2transmitted
Received Transition Y0Y1Y2received
Add Noise

Figure 5 General Data Flow Diagram

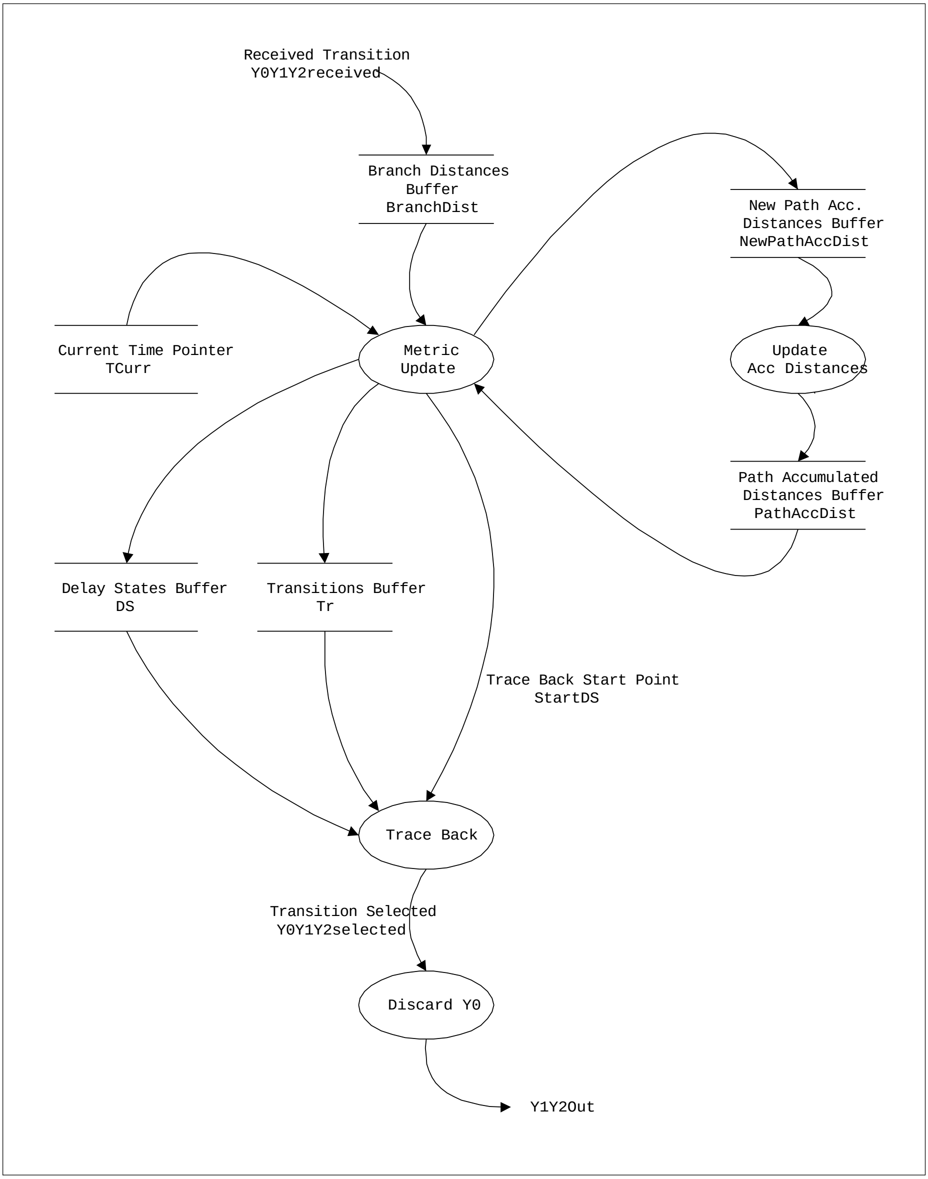

Received Transition
Y0Y1Y2received
Branch Distances Buffer BranchDist
New Path Acc. Distances Buffer NewPathAccDist
Current Time Pointer TCurr
Metric Update
Update Acc Distances
Path Accumulated Distances Buffer PathAccDist
Delay States Buffer DS
Transitions Buffer Tr
Trace Back Start Point StartDS
Trace Back
Transition Selected Y0Y1Y2selected
Discard Y0
Y1Y2Out

Figure 6 Viterbi Decoding Data Flow Diagram

3.2 Memory resources needed

Note: Note: As the smallest possible data type in C language, a char variable (coded with 8 bits) has been chosen to store a 2-bit symbol [Q1Q2], a 3-bit delay state [S0 S1 S2] as well as a 3-bit transition [Y0 Y1 Y2].

Input buffer (TabQ1Q2In)

This input buffer contains the successive input symbols $[Q1\ Q2]_{In}$ that have to be sent. The size for this linear buffer is defined by the parameter LENGTH_INPUT and can easily be modified.

<u>Size</u>: LENGTH_TB * 8 bits

Output buffer (TabQ1Q2Out)

The decoded output symbols $[Q1\ Q2]_{Out}$ calculated by the decoder are stored in this output buffer TabQ1Q2Out. It is a linear buffer and its size is the same than the one of the input buffer, since for each time period one input symbol is read and one output symbol is found and stored as well.

<u>Size</u>: LENGTH_TB * 8 bits

Differential Encoding Table (DiffEncod)

Knowing the past output $[Y1_{n-1}Y2_{n-1}]$ and the current input $[Q1_nQ2_n]$ of the differential encoder, this look-up table allows to compute the current output $[Y1_nY2_n]$ of the differential encoder, and thus to perform the differential coding.

Note: Please refer to the part 4.1 for more details about how to use this look-up table.

<u>Size</u>: 16 * 8 bits

Differential Decoding Table (DiffDecod)

Knowing the past input $[Y1_{n-1}Y2_{n-1}]$ and the current input $[Y1_nY2_n]$ of the differential decoder, this look-up table allows to compute the current output $[Q1_nQ2_n]$ of the differential decoder, and thus to perform the differential decoding.

Note: Please refer to the part 4.2 for more details about how to use this look-up table.

<u>Size</u>: 16 * 8 bits

Branch Distances buffer (BranchDist)

Since the received transitions $[Y0\ Y1\ Y2]_{\text{received}}$ an have been disturbed by channel noise, each transition value is considered as a possible representation of the received transition. Each one of the 8 possible transitions is a more or less likely representation of the received transition regarding its Hamming distance to it. This buffer contains the 8 distance values between the received transition $[Y0\ Y1\ Y2]_{\text{received}}$ and the 8 possible transitions [Y0 Y1 Y2].

Size: 32 * 8 bits

Path Accumulated Distances Buffers (PathAccDist and NewPathAccDist)

Since only one path, the most likely, is selected to lead to each delay state, only 8 paths have to be stored. Each path is identified by its accumulated distance which is the sum of the branch distances for each branch constituting the path. Since a new branch is added to each path at each symbol period, these accumulated path distances have to be updated each time interval.

Note: The previous accumulated path distances are needed until a new branch has been selected for all paths. Hence, it is not possible to directly replace the old accumulated path distances by the new ones. Therefore 2 buffers are required, PathAccDist and NewPathAccDist.

Size: The size of these 2 buffers depends on how many symbols are transmitted, i.e. on the parameter LENGTH_INPUT. In the worst case, one of the 8 paths stored is constituted of branches that have all a maximum cost of 3, and an initial cost of 16.

After LENGTH_INPUT time periods, the maximum accumulated distance is:

AccDistMax = 16 + LENGTH_INPUT * 3

- Currently: LENGTH_INPUT = 32

  AccDistMax = 112, which can be represented on 7 bits.

  We use 8 *8 bits to store the 8 accumulated distances for all paths.

- Best: Let us assume n is the number of bits necessary to represent the accumulate distance. Knowing the maximum path accumulated distance AccDistMax, n must confirm the following equation:

  $2^n \geq$ AccDistMax = 16 + LENGTH_INPUT * 3 and then:

  n ≥ Log (16 + 3* LENGTH_INPUT) / Log (2) (n ∈ N)

  8*n bits would be necessary, if the parameter LENGTH_INPUT is modified.

Metric Storage Circular Buffers (DS and Tr)

The decision which symbol has been transmitted after all symbols have been received. Therefore the whole path history has to be stored. Unfortunately, this can be very memory consuming. In addition, this leads often to an unacceptable delay of the decision. In practice, the path history is truncated to a smaller value, called LENGTH_TB in our application.

Since the decoder bases its decision on the path history of the previous LENGTH_TB-1 time periods, the metric storage buffers span LENGTH_TB time periods (including the current time period).

They are set up as circular buffers so that new branch information overwrites the oldest one at each time period.

To enable reconstruction of the 8 entire paths, 2 pieces of information have to be stored with respect to each selected branch at each time period:

- The transition that identify the branch is stored in the Tr circular buffer.
- The previous delay state from which the branch originates is stored in the DS circular buffer.

The format of these metric storage buffers is shown below, assuming the parameter LENGTH_TB is equal to 8 for instance:

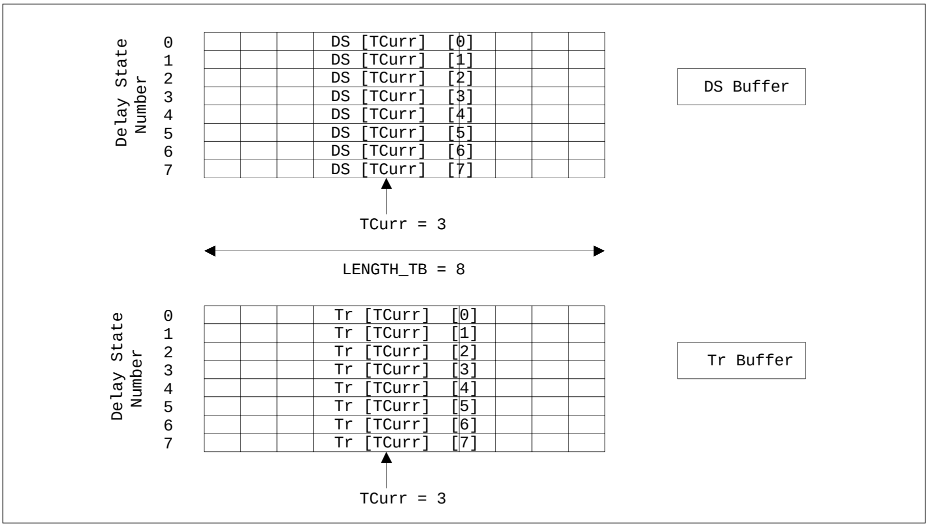


Figure 7 Metric Storage Buffers

Example: Referring to the expanded trellis diagram on figure 4, let us assume the current time period (identified by the pointer Tcurr) is 3 and that the current delay state is number 0 (called NewDS).

The Viterbi algorithm calculates the most likely branch leading to that delay state and finds for instance that this most likely branch is originating from the previous delay state [S0 S1 S2] number 2 (called PastDS), with a transition [Y0 Y1 Y2] equal to 3 (called Transition).

Then these 2 pieces of information are stored as follows:

DS [Tcurr] [NewDS] =PastDSi.e.DS [3] [0] = 2

Tr [Tcurr] [NewDS] = Transitioni.e.Tr [3] [0] = 3

Both buffers are set up as 8*LENGTH_TB symbol circular buffers, containing LENGTH_TB columns to represent a history of LENGTH_TB passes of the Viterbi algorithm. Each element of these 2 buffers is 3 bit wide.

<u>Size</u>: 8 * LENGTH_TB * 8 bits

4 Encoder Implementation

4.1 Differential Encoder

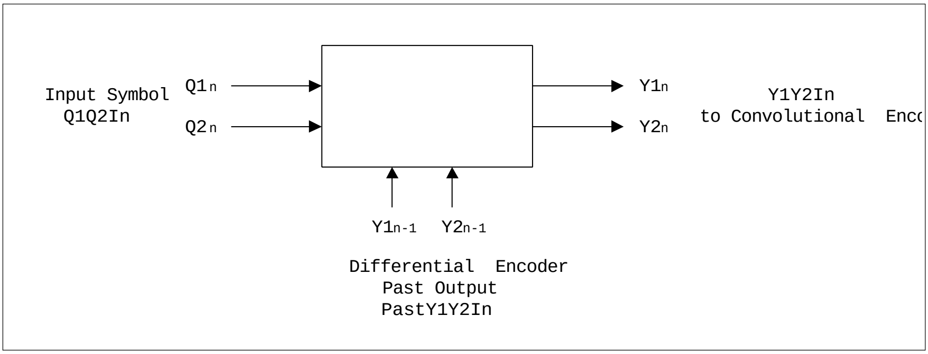


Figure 8 Differential Encoder Schematic

Implementing this differential encoder consists in storing the previous outputs of the differential encoder and then performing the appropriate EXCLUSIVE OR (^) and AND (•) functions defined by:

$$Y1_n = Q1_n \wedge Y1_{n-1}$$

$$Y2_n = (Q1_n \bullet Y1_{n-1}) \wedge Y2_{n-1} \wedge Q2_n$$

Chosen Implementation

Function Name: **DiffEncoding**

A table look up approach is taken to decrease the execution time of this routine. A 16-char table called DiffEncod is set up in memory. Each element of this table corresponds to a unique combination of bits [[$Y1_{n-1}Y2_{n-1}$] [$Q1_nQ2_n$]] and contains resulting differential encoding bits [$Y1_nY2_n$].

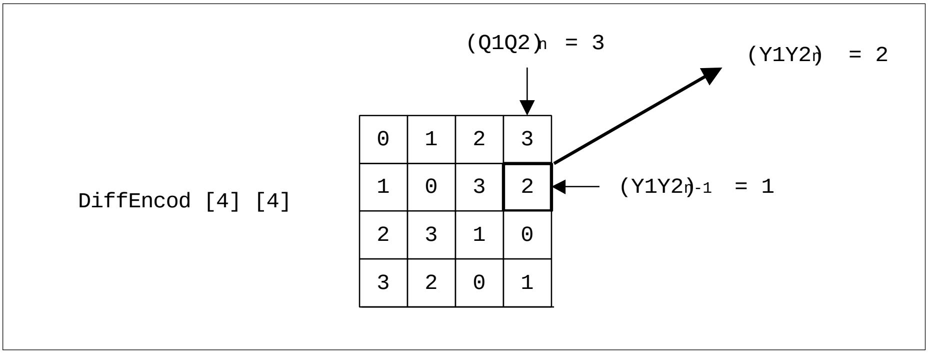


Figure 9 Differential Encoding look up table

Size of the table: 16 * 8 bits

Using: Knowing the past output $[Y1_{n-1}Y2_{n-1}]$ and the current input $[Q1_nQ2_n]$ of the encoder, the current output $[Y1_nY2_n]$ can easily be determined using the DiffEncod table and the following formula:

$[Y1_nY2_n]$ = DiffEncod $[Y1_{n-1}Y2_{n-1}]$ $[Q1_nQ2_n]$

Ex: Let us consider that:

- the past output is $[Y1_{n-1}Y2_{n-1}] = 2 \Leftrightarrow Y1_{n-1} = 1$ and $Y2_{n-1} = 0$
- the current input is $[Q1_nQ2_n] = 1 \Leftrightarrow Q1_n = 0$ and $Q2_n = 1$

Considering the logic formules leads to:

$Y1_n = Q1_n \wedge Y1_{n-1} = 0 \wedge 1 = 1$

$Y2_n = (Q1_n \bullet Y1_{n-1}) \wedge Y2_{n-1} \wedge Q2_n = (0 \bullet 1) \wedge 0 \wedge 1 = 0 \wedge 0 \wedge 1 = 1$

thus, $[Y1_nY2_n] = 3$

Using the DiffEncod table allows to come with this result too:

$[Y1_nY2_n]$ = DiffEncod $[Y1_{n-1}Y2_{n-1}]$ $[Q1_nQ2_n]$ = DiffEncod [2] [1] = 3

4.2 Convolutional Encoder

The 3 delays which compose the convolutional encoder are called S0, S1 and S2. The convolutional encoder takes the 2 differentially encoded bits $[Y1_nY2_n]$ and generates an output bit Y0. Y0 is often called the redundant bit because it carries only the forward error-correction information.

Detailled View

Functionally, the convolutional encoder is a 3-bit shift register interconnected by AND and XOR logic.

These 3 delays are referrred to as S0, S1 and S2 and represent the state of the encoder. Hence, the name of **delay states** to denote the state of the convolutional encoder.

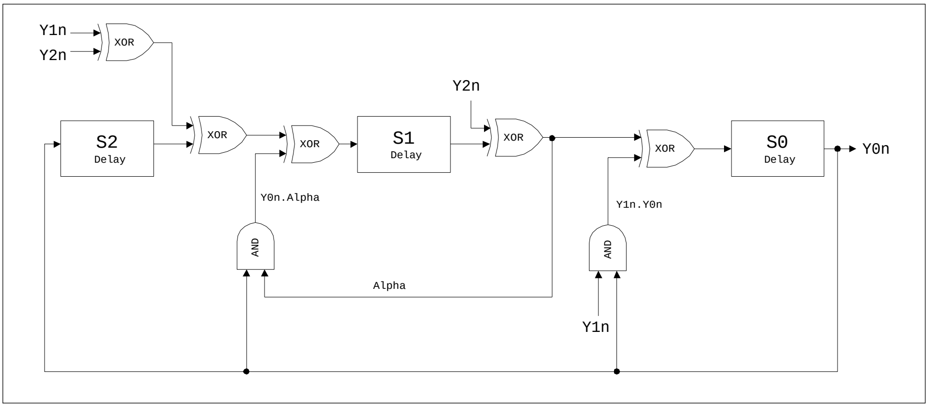


Figure 10 Convolutional Encoder

Chosen Implementation

Function Name: **ConvEncoding**

The convolutional encoder is implemented in the following way:

the content of the 3 delays - S0, S1 and S2 - is stored in a unique global variable called S0S1S2. Based on the configuration of the figure 10, the piece of information contained in each delay is used for each new symbol to determine the redundant bit Y0, and must be updated then. The output bit $Y0_n$ at each time period is the value of the delay 0 (S0) before it is updated.

4.3 Encoder Initialization

Function name: **InitEncoder**

This function is called to initialize both differential and convolutional encoders. To ensure that the encoding process begin with the null path - only composed of null transitions and always staying in the delay state number 0 -, following initializations have to be performed:

- The latest output of the differential encoder (called PastY1Y2In) must be set up on 0, otherwise it means that the preceding transmitted transition [Y0 Y1 Y2] (composed of the 2-bit signal PastY1Y2In plus a redundant bit Y0 added by the convolutional encoder) was not 0 and so did not belong to the null path.
- The initial state of the convolutional encoder - stored in the global variable called S0S1S2 - must be set up on 0, because staying in the null path means that the convolutional encoder stays in the delay state 0.

5 Channel Modelisation

The transmitted symbols $[Y0Y1Y2]_{transmited}$ are disturbed by a noise while transmitted over the channel. As a result, the received symbols $[Y0Y1Y2]_{received}$ can contain binary errors. The task of the decoder is then to correct the maximum of the disturbed bits.

This addition of noise is essential to test how resistant the decoder is when the channel disturbes the transmitted symbols.

Note: No assumption are made about any characteristic of the noise in this implementation.

Chosen implementation

Function Name: **AddNoise**

A relative simple approach is taken here. Each received symbol $[Y0Y1Y2]_{received}$ consists in an addition of one transmitted symbol $[Y0Y1Y2]_{transmitted}$ and a noise sample. The noise values are stored in the array called TabNoise, one noise value for each transmitted symbol.

Note: An ideal transmission without noise can be simulated by setting all the noise samples to the value zero.

6 Viterbi Decoding Implementation

6.1 Metric Initialization

Function name: **InitMetric**

This function is called to set up global metric variables.

In practice, it is important to assume that the 8 paths stored start from the delay state number 0. Therefore, DS and Tr arrays are reseted to zeros so that the "null path" is always chosen as the maximum-likelihood path at the beginning.

To ensure that the decoder always chooses branches that originate from delay state number 0 in the first time interval, the initial cost of the path originating from delay state number 0 is set to 0 whereas the rest of the paths are set to a greater cost, 16 for instance.

Then **PathAccDist** is initialized to the following array:

0
16
16
16
16
16
16
16

6.2 Viterbi Decoding – Dynamic Programming

Note: Each of the 5 following steps (from 6.2.1 to 6.2.5) must be performed for each time period.

6.2.1 Transition Receiving

The input to the Viterbi decoder is the 3 bit data stream $[Y0\ Y1\ Y2]_{received}$, which corresponds to a received transition (encoded symbol).

A new input transition is read every time period (or symbol period).

Note: A decision on the integrity of the input encoded symbol read will only be made by the decoder LENGTH_TB time periods later. This is the essence of Viterbi algorithm in which the actual decision is delayed until more information is available.

Note: A symbol input data is read from the array called TabInput defined in the main function. The size for this array is defined as parameter LENGTH_INPUT and can easily be modified. This symbol is then differential and convolutional encoded, and can be transmitted as a 3-bit transition at that time.

6.2.2 Hamming Branch Distance Calculation

Function Name: **ComputeBranchDistances**

The next step is to compute distance between the received transition and each of the 8 possible transitions [Y0 Y1 Y2].

The cost function is either Euclidean or Hamming distance, this application used the Hamming distance between the received encoded symbol and each possible transition, which is suitable for binary signals.

For each encoded symbol received, 8 distances to each transition are generated by the cost function, and stored in the array called BranchDist (means branch distances), as shown by the following example:

<u>Example</u>: SymbolIn = (011)

Distance { SymbolIn, (000) } = 2		**2**
Distance { SymbolIn, (001) } = 1		**1**
Distance { SymbolIn, (010) } = 1		**1**
Distance { SymbolIn, (011) } = 0	$\Rightarrow$ BranchDist =	**0**
Distance { SymbolIn, (100) } = 3		**3**
Distance { SymbolIn, (101) } = 2		**2**
Distance { SymbolIn, (110) } = 2		**2**
Distance { SymbolIn, (111) } = 1		**1**

6.2.3 Metric Update or Add–Compare–Select (ACS)

Function Name: **MetricUpdate**

The aim of this step is to find the current most likely branch to extend each path, and to compute the new accumulated cost for each path as following:

- Determine the 4 past delay states reaching the current delay state

- Add current branch distances to the path accumulated distances for each transition
- Comparison of the 4 input branches and selection of the survivor path
- Find the beginning point for the trace back

Note: Each of these 4 steps must be performed for each delay state.

Note: Each path can be identified by the current delay state it leads to. That is why we speak of accumulated cost for a delay state as well as a cost for a path.

Note: Most of calculation time is spent in this MetricUpdate procedure. This implementation is written with the aim of optimized speed while the code length is not so important. Efforts have to be made here to find the best solution regarding CPU cycles consuming. (see section 6.2.3.5)

6.2.3.1 Determine the 4 Past Delay States reaching the current Delay State

Close analysis of the V.32 trellis in Figure 4 reveals that there are a limited number of transitions (four) leading to each new delay state from the previous time period. The following table identifies the combination of previous delay states and transitions to reach each delay state for the current time period.

New Even DS [S0 S1 S2]	Past DS [S0 S1 S2]	Transition [Y0 Y1 Y2]
0	0	0
	1	2
	2	3
	3	1
2	0	2
	1	0
	2	1
	3	3
4	0	3
	1	1
	2	0
	3	2
6	0	1
	1	3
	2	2
	3	0

New Odd DS [S0 S1 S2]	Past DS [S0 S1 S2]	Transition [Y0 Y1 Y2]
1	4	4
	5	7
	6	6
	7	5
3	4	7
	5	4
	6	5
	7	6
5	4	5
	5	6
	6	7
	7	4
7	4	6
	5	5
	6	4
	7	7

Figure 11 Trellis singularities

Notice that all even-numbered delay states of the current time interval have links to the first 4 delay states of the previous time interval, whereas all odd-numbered new delay states have links to the last 4 past delay states.
So **it is relatively simple to process even- and odd-numbered delay states in two groups.**

Furthermore even-numbered delay states can be reached only by the first 4 transitions, whereas odd-numbered delay states can be reached only by the last 4 transitions.

All these trellis singularities can lead to an implementation with a look-up table to determinate the 4 possible previous delay states reaching the current delay state (NewDS).

So to favour the speed of the algorithm, trellis singularities have not been used in this implementation. Indeed, determining the 4 past delay states reaching the current delay state is time consuming and can be disregard by using a loop-unrolling. (See section 6.2.3.5)

6.2.3.2 Add current Branch Distances to the Path Accumulated Distances for each possible Transition

As shown before, each current delay state is linked to 4 previous delay states by 4 different branches. Because each current delay state is the target of 4 different paths origin from 4 previous delay states, the accumulated distance must be calculated for each of these 4 paths.

Each delay state is considered sequentially and the total metrics for each of the 4 possible paths leading to the current delay state are being calculated.

The following figure shows the possible transitions leading to delay state 0 for the V.32 trellis:

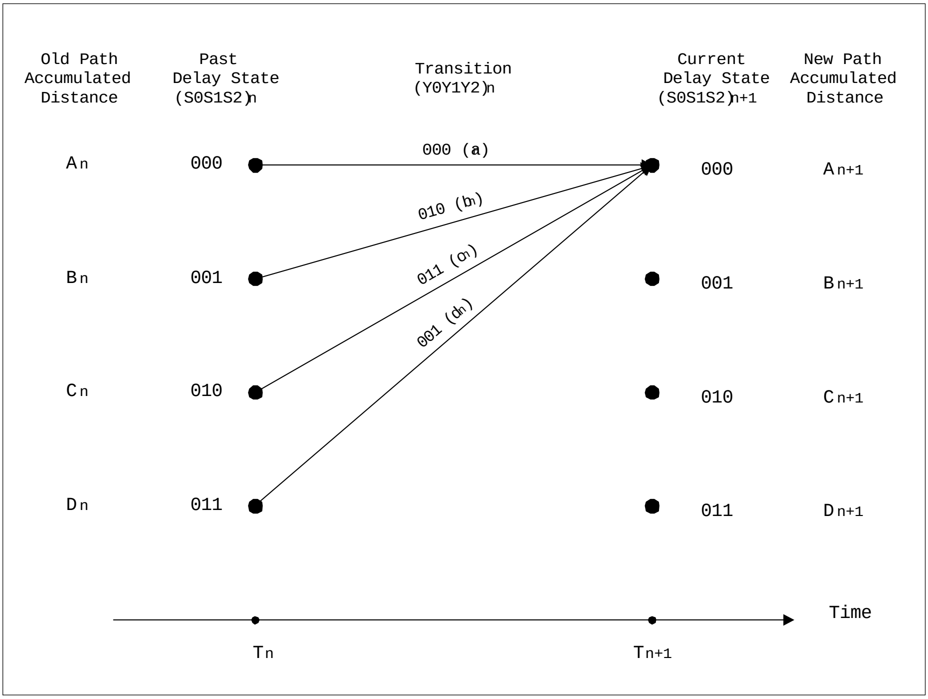

Old Path Accumulated Distance
Past Delay State (S0S1S2)n
Transition (Y0Y1Y2)n
Current Delay State (S0S1S2)n+1
New Path Accumulated Distance
An
Bn
Cn
Dn
000
001
010
011
000 (a)
010 (bn)
011 (cn)
001 (dn)
An+1
Bn+1
Cn+1
Dn+1
Time
Tn
Tn+1

Figure 12

- **Branch Distances:**

(a_n), (b_n), (c_n), (d_n) represent the branch distances, i.e. the Hamming distances between the input symbol and the different possible transitions.

These branch distances are being stored in the array called BranchDist.

- **Old Path Accumulated Distances:**

Accumulated distances for each delay state till the current time period (i.e. for each path stored) are stored in the array PathAccDist.

In this example, the 4 first values in this array are A_n, B_n, C_n and D_n.

- **Total Distances:**

On the preceding example, accumulating the total distance for each possible path will lead to these 4 total distance values:

$\{A_n + a_n\}$, $\{B_n + b_n\}$, $\{C_n + c_n\}$, $\{D_n + d_n\}$

Note: Overflow problem

As underlined when presenting the buffers PathAccDist and NewAccDist, it is impossible to continue to accumulate these distances without running into an overflow problem.

In this implementation, path accumulated distances are stored in the buffer PathAccDist composed of int variables, coded on 16 bits. It means that accumulated distance values stored must not exceed 65535.

The worst case would appear with a path whose transitions have always a maximum cost of 3 and which originates from a delay state whose initial cost is set up to 16. After (65535 – 16) / 3 < 21840 time periods, an overflow problem may occur.

Thus, **the number of input symbols to process must be lower than 2184**0.

LENGTH_INPUT < 21840 to avoid overflow

6.2.3.3 Comparison of the 4 Input Branches and Selection of the Survivor .... Path

The Viterbi algorithm now chooses the branch belonging to the maximum-likelihood path leading to the current delay state. It becomes the new fragment of the path reaching the current delay state.

The branch with the minimum total distance is selected as the most probable one, whereas the 3 others are discarded.

Note: For each one of the 8 delay states and for each time period, only one branch is selected and stored. It means that at any time only 8 paths are being stored in memory, each one leading to a different delay state. These 8 stored paths are called the ***survivor paths****.*

Example: Refering to the preceding example, let us assume $\{C_n + c_n\}$ is the minimum total distance value, that is to say the new accumulated distance A_{n+1} for delay state number 0 is $\{C_n + c_n\}$.

To enable reconstruction of the delay state sequence from a later point, the **following information needs to be stored** once the minimum distance branch is found:

- new accumulated distance for the current delay state

This new accumulated distance ($A_{n+1} = C_n + c_n$ here) can not be directly stored in the PathAccDist array, since the old accumulated distances (stored in PathAccDist) are being used to select the survivor path also for the others delay states till the end of this step 4.2.3.3.

Hence, the new accumulated distance for each new delay state is temporary stored in the array called NewPathAccDist.

- delay state of the previous time interval linked to the current delay

This past delay state number state (Past DS number 2 in the example) is being stored in the array DS, in the column pointed to by the current time pointer (Tcurr) and in the row corresponding to the current delay state (New DS number 0).

Hence the following formula:DS [TCurr] [NewDS] = Past DS

- transition that identify the branch selected

This transition (011 in our example) is being stored in the array Tr, in the column pointed to by the current time pointer (Tcurr) and in the row corresponding to the current delay state (New DS number 0 in the example).

Hence the following formula:Tr [TCurr] [NewDS] = Transition

This is the **metric update** that is repeated for each delay state, and for each time period. It is also called the **add – compare – select (ACS) operatio**n: accumulation of distance data, comparison of input branches, and selection of the maximum likelihood path.

6.2.3.4 Find the begining point for the Trace Back

The smallest path accumulated distance must be found and stored in the variable PathAccDistMin first.

Then, the path reaching the delay state associated with this smallest accumulated distance is considered as the most likely one and selected to receive output at the current time period. This minimum accumulated distance delay state is stored in the variable NewDS and will be used as the initial point to perform the trace back operation.

6.2.3.5 Chosen implementation for the Metric Update

The chosen implementation is optimized in relation to CPU cycles consumption. Delay states are processed one after each other without any loop, each delay state described by its own C code so as to avoid related calculations that are really time consuming (adress calculations for example) and encountered when using a loop.

Therefore, the C code is less dense (calculation for a path total distance is written 32 times, one calculation per existing branch in the trellis) but the generated assembler code is clearly faster.

6.2.4 Update Path Accumulated Distance Values

Function Name: **UpdateAccDistances**

Once the least-cost branches to the 8 delay states are identified and stored in appropriate tables, the path accumulated distance buffer PathAccDist can be updated with new accumulated distances (temporary saved in the NewPathAccDist array).

Note: this update routine is written directly in TriCore Assembler for a minimal time consumption. TriCore assembler commands are used and enable to load and store 32 bit-data in only one CPU cycle (optimal using of the TriCore 32 bit Architecture).

6.2.5 Trace back

Function Name: **TraceBack**

The purpose of the trace back routine is to determine the most likely transmitted transition by tracing back the maximum-likelihood path through the trellis once it has been identified. For every time period, accumulated distances to each delay state have been calculated (6.2.3.2).

Furthermore, the minimum-distance branch (identified by a transition [Y0 Y1 Y2]) to each delay state has been stored as well as the delay state it came from [S0 S1 S2] in metric storage buffers DS and Tr respectively (6.2.3.3). Storing this data during the last LENGTH_TB time periods creates a history, making it possible to trace back along the most likely path to get the most likely output of the decoder.

6.2.5.1 Evaluation of the Convolutional Encoder State

A loop is used to trace back history of the path chosen as the most probable one through LENGTH_TB – 1 time periods.

Each cycle of this loop corresponds to a trace back time period and is identified by the T_TB pointer (Time period of the trace back). This pointer is initialized to the current time period value Tcurr and will be decremented at each cycle.

The following processing needs to be performed during one loop cycle:

- Find the delay state from which the current delay state NewDS comes, and store it in the variable PastDS.
- This preceding delay state (PastDS) becomes the current delay state (NewDS) for the next loop cycle.
- Trace back time period pointer (T_TB) must also be decremented.
- Remembering the metric storage buffers DS and TR are circular, T_TB pointer can not be assigned to a negative value. If T_TB value becomes equal to -1 after decrementing, it must be reseted to the value LENGTH_TB - 1 so that it points to the preceding entry.

At the end of the loop iterations, the oldest delay state [S0S1S2] will be found and stored in the NewDS variable: it determines the most likely state of the convolutional encoder LENGTH_TB – 1 time periods backward.

6.2.5.2 Find the most probable transition transmitted

By means of the most likely delay state detected at the end of the trace back, we can retrieve the respective transition [Y0Y1Y2] from the array Tr.

The transition (stored in the buffer Tr) taken to get to that most likely delay state (stored in the variable NewDS) is selected by the Viterbi convolutional decoder as the most probable transition transmitted by the encoder.

Note: The output for the current time period (TCurr) reflects a decision made by the decoder on symbols received up to LENGTH_TB time periods later. This means **that the output symbol is necessarily delayed by LENGTH_TB time periods in relation with input symbo**l.

6.2.5.3 Discard YO

The most significant bit (Y0) of the transition [Y0Y1Y2] selected by the decoder can be stripped off at this point since it is only a redundant bit added during the encoding process. Then the resulting 2-bit differential encoded symbol Y1Y2Out is the output of the Viterbi convolutional decoder for the current time period TCurr.

7 Differential Decoding Implementation

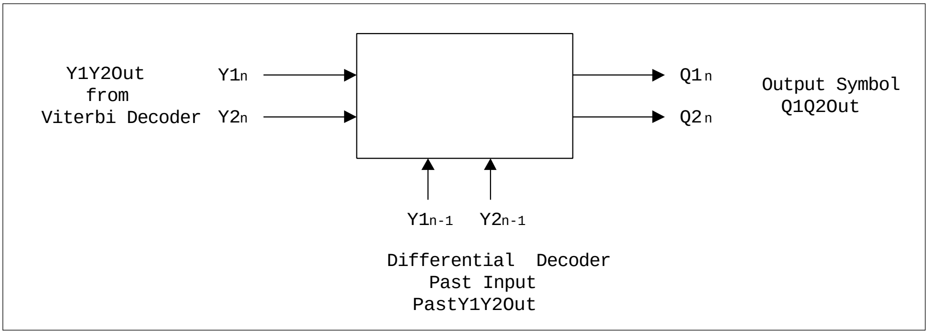


Figure 13 Differential Decoder Schematic

Implementing this differential decoder consists in storing the previous inputs of the differential decoder and then performing the appropriate EXCLUSIVE OR (^) and AND (•) functions defined by:

$Q1_n = Y1_n \wedge Y1_{n-1}$

$Q2_n = (Q1_n \bullet Y1_{n-1}) \wedge Y2_{n-1} \wedge Y2_n$

7.1 Chosen implementation

Function Name: **DiffDecoding**

A table look up approach is taken to decrease the execution time of this routine. A 16-char table called DiffDecod is set up in memory. Each element of this table corresponds to a unique combination of bits [[$Y1_{n-1}Y2_{n-1}$] [$Y1_nY2_n$]] and it contains resulting differential encoding bits [$Q1_nQ2_n$].

Size of the table: 16 * 8 bits

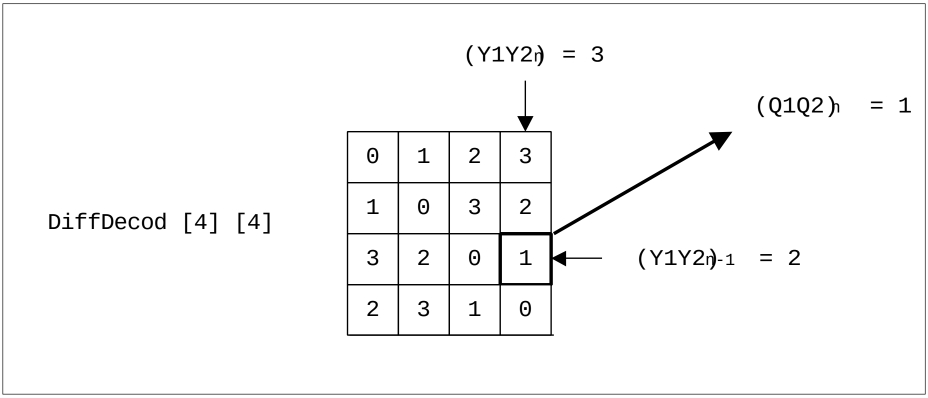


Figure 14 Differential Decoding look up table

Using: Knowing the past input [$Y1_{n-1}Y2_{n-1}$] and the current input [$Y1_nY2_n$] of the differential decoder, the current output [$Q1_nQ2_n$] can easily be determined using the DiffDecod table and the following formula:

[$Q1_nQ2_n$] = DiffDecod [$Y1_{n-1}Y2_{n-1}$] [$Y1_nY2_n$]

Example: Let us consider that:

- the past input is [$Y1_{n-1}Y2_{n-1}$] = 3 ⇔ $Y1_{n-1}$ = 1 and $Y2_{n-1}$ = 1
- the current input is [$Y1_nY2_n$] = 2 ⇔ $Y1_n$ = 1 and $Y2_n$ =0

Considering the preceding logic formules leads to:

$Q1_n = Y1_n \wedge Y1_{n-1} = 1 \wedge 1 = 0$

$Q2_n = (Q1_n \bullet Y1_{n-1}) \wedge Y2_{n-1} \wedge Y2_n = (0 \bullet 1) \wedge 1 \wedge 0 = 0 \wedge 1 \wedge 0 = 1$

thus, [$Q1_nQ2_n$] = 1

Using the DiffDecod table allows to come with this result too:

[$Q1_nQ2_n$] = DiffDecod [$Y1_{n-1}Y2_{n-1}$] [$Y1_nY2_n$] = DiffDecod [3] [2] = 1

7.2 Differential Decoder Initialization

Function name: **InitDecoder**

This function is called to initialize the differential decoder. The decoder also wait for a succession of zeros - succession of transitions composing the null path - to begin with its decoding task. It is therefore assumed that the initial latest input of the differential decoder (called PastY1Y2Out) is O too.

8 Optimization Strategy

The TASKING C cross-compiler (ctri) allows the user to control the special functions of the TriCore in C with extensions to the C language.

The following C extensions are used in this implementation:

- _near storage type:
  Using the _near addressing qualifier, allows the compiler to generate faster access code for frequently used variables. The data object is directly addressable using the absolute addressing mode.
- inline C functions:
  The _inline keyword is used to signal the compiler to inline the function body instead of calling the function.

  Note: The debugger cannot step-into an inline function.
- inline assembly:
  ctri supports inline assembly. Writing a function directly in assembler is especially useful when the compiler generates non optimized assembler code. In this implementation, the function UpdateAccDist is written using inline assembler.
- static storage specifier:
  using this keyword with a variable that is local to a function allows the last value of the variable to be preserved between successive calls to that function.

9 Results Presentation

A global constant called OUTPUT_FILES is set up to specify if output files must be generated or not. If OUTPUT_FILES is set up to 1 then two output files containing all simulation results are generated as decribed further.

9.1 Printing Results to the Screen

If the global constant OUTPUT_FILES is initialized to another value than 1, then simulations results are only printed to the screen.

Only input symbols (read from the buffer TabInput) and output symbols (stored in the buffer TabOuput) are both printed to the screen, so as to compare them and to see how performant the decoder is.

This configuration is used to benchmark the decoding. (no time wasting in opening and writing into output files).

9.2 Printing Results to Files

This second configuration (OUTPUT_FILES initialized to 1) make use of output text files to print all the intermediate results from the input symbols to the decoded output symbols.

Two output text files called ‘ transmission.txt’ and ‘ reception.txt’ are generated.

The aim of this configuration is especially to be able to reconstruct the path selected by the decoder through the trellis.

Transmission File (transmission.txt)

This file gives information about all the necessary data to show the path taken by the encoder through the trellis and about the noise disturbing the transmission too.

In this file are stored the following data:

- Current time value: from 0 to 31 since the simulation with the test sequence consists of 32 symbols.
- Input symbol: read from the buffer TabInput, it represents the data $[Q1Q2]_{In}$ to be transmitted.
- Encoder State: it represents the current state of the convolutional encoder [S0S1S2].
- Transition: denotes the transition $[Y0Y1Y2]_{transmitted}$ calculated by the encoder and that is transmitted over the transmission channel.

- Noise: noise sample, read from the buffer TabNoise that disturbs the transmitted transition in the channel.
- Trans_Disturbed: transition resulting from the superposition of the transmitted transition and the noise sample. This disturbed transition represents the transition $[Y0Y1Y2]_{received}$ received by the decoder.

Time	Input	Encoder		Channel	
	Q1Q2	State S	Transition Y	Noise	Trans_ Disturbed
0	0	0	0	0	0
1	0	0	0	2	2
2	0	0	0	0	0
3	3	4	3	0	3
4	1	7	6	0	6
5	2	1	5	0	5
6	2	6	3	0	3
7	3	3	5	0	5
8	0	0	1	0	1
9	1	0	0	0	0
10	3	4	3	1	2
11	1	7	6	0	6
12	2	1	5	0	5
13	0	4	1	0	1
14	3	7	6	2	4
15	1	7	7	0	7
16	0	7	7	0	7
17	0	7	7	0	7
18	0	7	7	0	7
19	0	7	7	4	3
20	0	7	7	0	7
21	0	7	7	0	7
22	0	7	7	0	7
23	0	7	7	0	7
24	0	7	7	0	7
25	0	7	7	0	7
26	0	7	7	0	7
27	0	7	7	0	7
28	0	7	7	0	7
29	0	7	7	0	7
30	0	7	7	0	7
31	0	7	7	0	7

Figure 15 Transmission File for a test input sequence

Reception File (reception.txt)

This file gives information about all the necessary data to reconstruct the path selected by the decoder through the trellis as the most likely one.

In this file are stored the following data:

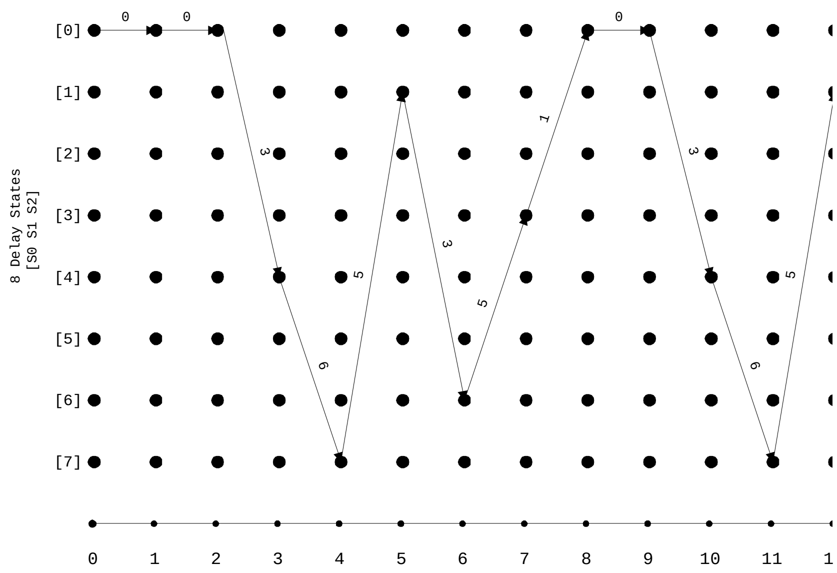
- Current time value: from 0 to 31 since the simulation with the test sequence consists of 32 symbols.
- Output symbol: stored in the buffer TabOutput, it represents the decoded symbol $[Q1Q2]_{Out}$.
- Evaluated Encoder State: it represents the state of the convolutional encoder [S0S1S2] evaluated by the decoder as the most likely one.
- Transition: denotes the transition $[Y0Y1Y2]_{selected}$ chosen by the decoder as the most probable one.

Time	Output	Encoder_Evaluation	
	Q1Q2	State S	Transition Y
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0
10	0	0	0
11	0	0	0
12	0	0	0
13	0	0	0
14	0	0	0
15	0	0	0
16	0	0	0
17	0	0	0
18	3	4	3
19	1	7	6
20	2	1	5
21	2	6	3
22	3	3	5
23	0	0	1
24	1	0	0
25	3	4	3
26	1	7	6

27	2	1	5
28	0	4	1
29	3	7	6
30	1	7	7
31	0	7	7

Figure 16 Reception File for the test input sequence

9.3 Most likely Path selected for the test input sequence



Transition Received Y0Y1Y2received	2	0	3	6	5	3	5	1	0	2	6	5
Transition Selected Y0Y1Y2selected	0	0	3	6	5	3	5	1	0	3	6	5

Bits corrected	1									1		

10 Appendix A - C Code Implementation for the Viterbi Algorithm

```
//*********************************************************************
// @ModuleViterbi Decoder
// @FilenameParameters.h
// @Project V.32 Modem Implementation
//---------------------------------------------------------------------
// @ControllerTriCore
//
// @CompilerTasking TriCore C Cross-Compiler
//
// @DescriptionThis module contains all specific parameters
// for the V.32 Decoding application
//
//---------------------------------------------------------------------
// @Date 19/02/99 15:00:00 AM
//
//*********************************************************************
//*********************************************************************
// @Defines
//*********************************************************************
#defineLENGTH_INPUT 32
#defineLENGTH_TB 16
#define OUTPUT_FILES 0/* If you want output files to be printed then put
the value of this constant to 1 */


//*********************************************************************
// @ModuleViterbi Decoder
// @FilenamePrototypes.h
// @Project V.32 Modem Implementation
//---------------------------------------------------------------------
// @ControllerTriCore
//
// @CompilerTASKING TriCore C Cross-Compiler
//
// @Description This file contains all function prototypes for the V.32
Encoding
// and Decoding
//
//---------------------------------------------------------------------
// @Date 19/02/99 15:00:00 AM
//
//*********************************************************************
```

Appendix A - C Code Implementation for the Viterbi Algorithm

```
//*****************************************************************
// @Prototypes of global functions
//*****************************************************************
void InitEncoder (void);
void InitDecoder (void);
void InitMetric (void);
char DiffEncoding (char Q1Q2In);
char ConvEncoding (char Y1Y2In);
char AddNoise (char Y0Y1Y2_transmitted, char Noise);
_inline
void    ComputeBranchDistances    (char    Y0Y1Y2_transmitted,    char
BranchDist[8]);
_inline
char MetricUpdate (char *BranchDist, int *NewPathAccDist);
_inline
char TraceBack (char StartDS);
void UpdateAccDistances (int PathAccDist[8], int NewPathAccDist[8]);
char ViterbiDecoding (char Y0Y1Y2_received);
char DiffDecoding (char Y1Y2Out);
void IncrementTimePeriod (void);



//*****************************************************************
// @ModuleViterbi Decoder
// @FilenameSource.c
// @Project V.32 Modem Implementation
//-----------------------------------------------------------------
// @ControllerTriCore
//
// @CompilerTASKING TriCore C Cross-Compiler
//
// @DescriptionThis file contains all the functions both to encode and
decode
// the data in accordance with V.32 specifications
//
//-----------------------------------------------------------------
// @Date 19/02/99 15:00:00 AM
//
//*****************************************************************



//*****************************************************************
// @Project and Libraries Includes
//*****************************************************************
```

Appendix A - C Code Implementation for the Viterbi Algorithm

```
#include <stdio.h> /* Prototype for the function "printf"*/
#include "Parameters.h"/* File containing all the constants used */
#include "Prototypes.h" /* File containing the prototypes for all the
functions used */
//*******************************************************************
// @Global Variables
//*******************************************************************
_near static char DS[LENGTH_TB][8], Tr[LENGTH_TB][8], TCurr;
_near static int PathAccDist[8];
char StateEncoderEval; /* Convolutional encoder state evaluated by the
decoder*/
char  TransitionEval;  /*  most  likely  transition  transmitted  by  the
encoder evaluated
by the decoder*/
_near static char PastY1Y2In, PastY1Y2Out, S0S1S2;
_near static char DiffEncod[4][4] = { 0, 1, 2, 3,
1, 0, 3, 2,
2, 3, 1, 0,
3, 2, 0, 1};
_near static char DiffDecod[4][4] = { 0, 1, 2, 3,
1, 0, 3, 2,
3, 2, 0, 1,
2, 3, 1, 0};



//*******************************************************************
// @Function void main (void)//
//-------------------------------------------------------------------
// @DescriptionMain function
//
//-------------------------------------------------------------------
// @Returnvaluenone
//
//-------------------------------------------------------------------
// @Parametersnone
//
//-------------------------------------------------------------------
// @Date 19/02/99 15:00:00 AM
//
//*******************************************************************
void main (void)
{
FILE *pf_transmission;
FILE *pf_reception;
int SymbolCount;
char Q1Q2In, Q1Q2Out;
char Noise;
char Y1Y2In, Y0Y1Y2_transmitted;
char Y1Y2Out, Y0Y1Y2_received;
char TabQ1Q2In[LENGTH_INPUT] = {0,0,0,3,1,2,2,3,0,1,3,1,2,0,3,1,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
char TabNoise[LENGTH_INPUT] = {0,2,0,0,0,0,0,0,0,0,1,0,0,0,2,0,
0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0};
char TabQ1Q2Out[LENGTH_INPUT];
/* print the output files headers if output files are needed*/
if (OUTPUT_FILES == 1)
{pf_transmission = fopen( "transmission.txt","w");
fprintf(pf_transmission,"\n Time | Input | Encoder | Channel");
fprintf(pf_transmission,"\n  |  Q1Q2  |  State  S  Transition  Y  |  Noise
Trans_Disturbed");
fprintf(pf_transmission,"\n-------------------------------------------------
---------------------");
pf_reception = fopen( "reception.txt","w");
fprintf(pf_reception,"\n Time | Output | Encoder_Evaluati on");
fprintf(pf_reception,"\n | Q1Q2 | State S Transition Y");
fprintf(pf_reception,"\n--------------------------------------------------
");
}
/* Encoder Initialization */
InitEncoder ();
/* Decoder Initialization */
InitDecoder();
/* Metric Initialization */
InitMetric();
printf("\n---------------------------------\n");
for (SymbolCount = 0; SymbolCount < LENGTH_INPUT; SymbolCount++)
{
/* Read the current input symbol from the input buffer TabQ1Q2In */
Q1Q2In = TabQ1Q2In[SymbolCount];
if (OUTPUT_FILES == 1)
{fprintf(pf_transmission,"\n %2d | %2d | ",SymbolCount,Q1Q2In);}
/* Perform the Differential Encoding */
Y1Y2In = DiffEncoding (Q1Q2In);
/* Perform the Convolutional Encoding */
Y0Y1Y2_transmitted = ConvEncoding (Y1Y2In);
if (OUTPUT_FILES == 1)
{fprintf(pf_transmission,"%2d %2d ",S0S1S2, Y0Y1Y2_transmitted);}
/*  Channel  simulation:  add  some  errors  on  the  transmitted  transition
Y0Y1Y2_transmitted:
Y0Y1Y2_received = Y0Y1Y2_transmitted + Noise */
Noise = TabNoise[SymbolCount];
Y0Y1Y2_received = AddNoise (Y0Y1Y2_transmitted, Noise);
if (OUTPUT_FILES == 1)
```

Appendix A - C Code Implementation for the Viterbi Algorithm

```
{fprintf(pf_transmission,"| %2d %2d",Noise,Y0Y1Y2_transmitted );}
/* Perform the Convolutional Decoding: Run the Viterbi algorithm
to estimate the transmitted transition */
if (SymbolCount == (LENGTH_TB+1))
{printf("Time to mesure...\n");}
Y1Y2Out= ViterbiDecoding (Y0Y1Y2_received);
/* Perform the Differential Decoding */
Q1Q2Out = DiffDecoding (Y1Y2Out);
/* Write the current Output Symbol in the Output Buffer TabQ1Q2Out */
TabQ1Q2Out[SymbolCount] = Q1Q2Out;
if (OUTPUT_FILES == 1)
{fprintf(pf_reception,"\n %2d | %2d | %2d %2d",
SymbolCount,Q1Q2Out,StateEncoderEval, TransitionEval);}
/* Print both Input and Output Symbols */
if (SymbolCount >= 16)
printf("%d %d\n",TabQ1Q2In[SymbolCount - 15], Q1Q2Out);
/* Increment the current Time Pointer */
IncrementTimePeriod();
}
printf("\n\n");
if (OUTPUT_FILES == 1)
{fclose(pf_reception);
fclose(pf_transmission);}
}



//*******************************************************************
// @Function void InitMetric (void)
//
//-------------------------------------------------------------------
// @DescriptionThis function initializes the entire arrays DS and Tr to
zeros so that
// the "null path" is always chosen as the maximum-likelihood path at
// the beginning.
//
// To ensure that the decoder always chooses branches that
// originate from delay state number 0 in the first time interval,
// the initial cost of the path originating from delay state number
// 0 is set to 0 whereas the rest of the paths are set to a greater
// cost, 16 for instance.
//
// Current time pointer Tcurr is initialized to 0.
//
//-------------------------------------------------------------------
// @Returnvaluenone
//
```

Appendix A - C Code Implementation for the Viterbi Algorithm

```
//-------------------------------------------------------------------
// @Parametersnone
//
//-------------------------------------------------------------------
// @Date 19/02/99 15:00:00 AM
//
//*******************************************************************
void InitMetric ()
/* Initialization of the metric so that the "null path" is always chosen
as the most likely one at the start */
{
int i, j;
TCurr = 0;
PathAccDist[0] = 0;
for (i = 1; i < 8; i++)
PathAccDist[i] = 16;
for (j = 0; j < LENGTH_TB; j++)
for (i = 0; i < 8; i++)
{
DS[j][i] = 0;
Tr[j][i] = 0;
}
}



//*******************************************************************
// @Function void InitEncoder (void)
//
//-------------------------------------------------------------------
// @DescriptionThis function initializes both differential encoder and
convol utional
// encoder so that the initial state of the encoder is the null path
//
//-------------------------------------------------------------------
// @Returnvaluenone
//
//-------------------------------------------------------------------
// @Parametersnone
//
//-------------------------------------------------------------------
// @Date 19/02/99 15:00:00 AM
//
//*******************************************************************
void InitEncoder ()
{PastY1Y2In = 0;
S0S1S2 = 0;
}


//*******************************************************************
// @Function void InitDecoder (void)
//
//-------------------------------------------------------------------
// @DescriptionThis function initializes the latest input of the encoder
to zero
// so that the "null path" is always chosen as the maximum-likelihood
// path at the beginning.
//
//
//
//-------------------------------------------------------------------
// @Returnvaluenone
//
//-------------------------------------------------------------------
// @Parametersnone
//
//-------------------------------------------------------------------
// @Date 19/02/99 15:00:00 AM
//
//*******************************************************************
void InitDecoder ()
{PastY1Y2Out = 0;
}



//*******************************************************************
// @Function char DiffEncoding (char Q1Q2In)
//
//-------------------------------------------------------------------
//  @DescriptionThis  function  differentially  encodes  the  input  symbol
Q1Q2In
// in a 2-bit encoded data Y1Y2In
//
//-------------------------------------------------------------------
// @ReturnvalueThe differential 2-bit encoded data Y1Y2In
//
//-------------------------------------------------------------------
// @ParametersQ1Q2In: the input symbol that has to be transmitted
//
//-------------------------------------------------------------------
// @Date 19/02/99 15:00:00 AM
```

Appendix A - C Code Implementation for the Viterbi Algorithm

```
//
//*******************************************************************
char DiffEncoding (char Q1Q2In)
{
char NewY1Y2In;
NewY1Y2In = DiffEncod[PastY1Y2In][Q1Q2In];
PastY1Y2In = NewY1Y2In;
return (NewY1Y2In);
}



//*******************************************************************
// @Function char ConvEncoding (char Y1Y2In))
//
//-------------------------------------------------------------------
//   @DescriptionThis   function   convolutionaly   encodes   the   2-bit
differentially
// encoded data Q1Q2In in a 3-bit transition Y0Y1Y2transmitted
//
//-------------------------------------------------------------------
//   @Returnvalue   the   3-bit   transition   Y0Y1Y2transmitted   that   si
transmitted over
// the channel
//
//-------------------------------------------------------------------
// @ParametersY1Y2In: the 2-bit differentially encoded data
//
//-------------------------------------------------------------------
// @Date 19/02/99 15:00:00 AM
//
//*******************************************************************
char ConvEncoding (char Y1Y2In)
{
char NewY0Y1Y2 =Y1Y2In;
/* NewY0Y1Y2 = [Y0 Y1 Y2], based on the 2 bit input [Y1 Y2]
with a third redundant bit Y0 computed in this function
It is initialised to the value of Y1Y2 and will be increased
of 4 if Y0 is equal to 1 */
char NewY0, NewY1, NewY2;
char S0, S1, S2;
char NewS0, NewS1, NewS2;
char alpha;
S0 = S0S1S2 >> 2;
S1 = (S0S1S2 & 2) >> 1;
S2 = S0S1S2 & 1;
NewY0 = S0;
```

Appendix A - C Code Implementation for the Viterbi Algorithm

```
NewY1 = (Y1Y2In & 2) >> 1;
NewY2 =Y1Y2In & 1;
alpha = S1 ^ NewY2;
NewS0 = alpha ^(S0 & NewY1);
NewS1 = (S2 ^ (NewY1 ^ NewY2)) ^ (alpha & S0);
NewS2 = S0;
if (NewY0 == 1)
{NewY0Y1Y2 = NewY0Y1Y2 + 4;
}
/* Update encoder state */
S0S1S2 = NewS0*4 + NewS1*2 + NewS2;
return (NewY0Y1Y2);
}



//*******************************************************************
// @Function char AddNoise (char Y0Y1Y2_transmitted, char Noise)
//
//-------------------------------------------------------------------
//  @Description  This  function  simulate  the  transmission  over  a
disturbing channel
//
//-------------------------------------------------------------------
// @Returnvalue the disturbed transition Y0Y1Y2_received
//
//-------------------------------------------------------------------
// @ParametersY0Y1Y2_transmitted: the transmitted transition
//
//-------------------------------------------------------------------
// @Date 19/02/99 15:00:00 AM
//
//*******************************************************************
char AddNoise (char Y0Y1Y2_transmitted, char Noise)
{
return(Y0Y1Y2_transmitted ^ Noise);
}



//*******************************************************************
// @Function _inline
// void ComputeBranchDistances (char Y0Y1Y2_received,
// char BranchDist[8])
//
//-------------------------------------------------------------------
```

Appendix A - C Code Implementation for the Viterbi Algorithm

```
// @DescriptionThis function computes Hamming distance values between
the
// current transition received and the 8 possible transition values,
// and stores these distances in the array BranchDist
//
//--------------------------------------------------------------------
// @Returnvaluenone
//
//--------------------------------------------------------------------
// @Parameters char Y0Y1Y2_received: transition received by the decoder
//
// BranchDist[8]: 8 distance values between the current received
// transition and the 8 possible transition values
//
//--------------------------------------------------------------------
// @Date 19/02/99 15:00:00 AM
//
//********************************************************************
_inline
void ComputeBranchDistances (char Y0Y1Y2_received, char BranchDist[8])
{
BranchDist[Y0Y1Y2_received] = 0;
BranchDist[1^Y0Y1Y2_received] = 1;
BranchDist[2^Y0Y1Y2_received] = 1;
BranchDist[3^Y0Y1Y2_received] = 2;
BranchDist[4^Y0Y1Y2_received] = 1;
BranchDist[5^Y0Y1Y2_received] = 2;
BranchDist[6^Y0Y1Y2_received] = 2;
BranchDist[7^Y0Y1Y2_received] = 3;
};



//********************************************************************
// @Function _inline
// char MetricUpdate ( char *BranchDist, int *NewPathAccDist)
//
//--------------------------------------------------------------------
// @DescriptionThis function finds the 8 most likely branches to extend
each path
// stored in memory and computes the new accumulated distances for
// each one.
//
//--------------------------------------------------------------------
// @Returnvaluedelay state chosen as the start point for the trace back
operation
//
```

Appendix A - C Code Implementation for the Viterbi Algorithm

```
//-------------------------------------------------------------------
//  @ParametersPathAccDist[8]:  buffer  containing  accumulated  distances
for
// each one of the 8 paths stored
//
// BranchDist[8]: buffer containing the 8 distances between the
// received transition and each possible transition
//
// NewPathAccDist[8]: buffer where the 8 new accumulated distances are
// being stored during the current time period, after
// the 8 most likely branches have been chosen to
// extend the 8 paths
//
//-------------------------------------------------------------------
// @Date 19/02/99 15:00:00 AM
//
//*******************************************************************
_inline
char MetricUpdate ( char *BranchDist, int *NewPathAccDist)
{
int TotalDist, PathAccDistMin;
char BestDS;
/* NewDS with the smallest accumulated distance, which will be used as
the start point
for the trace back operation */
int GlobalDistMin;
/* global minimum path accumulated Distance between all the 8 existing
path accumulated distances (one for each DS) The corresponding NewDS -
called BestDS- will be the start point for the trace back and is the
output char of the function */

/* NewDS 0 */
/* PastDS 0, Transition 0 */
PathAccDistMin = PathAccD ist[0] + BranchDist[0];
DS[TCurr][0] = 0;
Tr[TCurr][0] = 0;
/* PastDS 1, Transition 2 */
TotalDist = PathAccDist[1] + BranchDist[2];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][0] = 1;
Tr[TCurr][0] = 2;
}
/* PastDS 2, Transition 3 */
TotalDist = PathAccDist[2] + BranchDist[3];
if ( TotalDist < PathAccDistMin)
```

```
{
PathAccDistMin = TotalDist;
DS[TCurr][0] = 2;
Tr[TCurr][0] = 3;
}
/* PastDS 3, Transition 1 */
TotalDist = PathAccDist[3] + BranchDis t[1];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][0] = 3;
Tr[TCurr][0] = 1;
}
NewPathAccDist[0] = PathAccDistMin;
GlobalDistMin = PathAccDistMin;
BestDS = 0;
/* NewDS 1 */
/* PastDS 4, Transition 4 */
PathAccDistMin = PathAccDist[4] + BranchDist[4];
DS[TCurr][1] = 4;
Tr[TCurr][1] = 4;
/* PastDS 5, Transition 7 */
TotalDist = PathAccDist[5] + BranchDist[7];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][1] = 5;
Tr[TCurr][1] = 7;
}
/* PastDS 6, Transition 6 */
TotalDist = PathAccDist[6] + BranchDist[6];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][1] = 6;
Tr[TCurr][1] = 6;
}
/* PastDS 7, Transition 5 */
TotalDist = PathAccDist[7] + BranchDist[5];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][1] = 7;
Tr[TCurr][1] = 5;
}
NewPathAccDist[1] = PathAccDistMin;
if (NewPathAccDist[1] < GlobalDistMin)
```

```
{GlobalDistMin = NewPathAccDist [1];
BestDS = 1;
}
/* NewDS 2 */
/* PastDS 0, Transition 2 */
PathAccDistMin = PathAccDist[0] + BranchDist[2];
DS[TCurr][2] = 0;
Tr[TCurr][2] = 2;
/* PastDS 1, Transition 0 */
TotalDist = PathAccDist[1] + BranchDist[0];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][2] = 1;
Tr[TCurr][2] = 0;
}
/* PastDS 2, Transition 1 */
TotalDist = PathAccDist[2] + BranchDist[1];
if ( TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][2] = 2;
Tr[TCurr][2] = 1;
}
/* PastDS 3, Transition 3 */
TotalDist = PathAccDist[3] + BranchDist[3];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][2] = 3;
Tr[TCurr][2] = 3;
}
NewPathAccDist[2] = PathAccDistM in;
if (NewPathAccDist[2] < GlobalDistMin)
{GlobalDistMin = NewPathAccDist[2];
BestDS = 2;
}
/* NewDS 3 */
/* PastDS 4, Transition 7 */
PathAccDistMin = PathAccDist[4] + BranchDist[7];
DS[TCurr][3] = 4;
Tr[TCurr][3] = 7;
/* PastDS 5, Transition 4 */
TotalDist = PathAccDist[5] + BranchDist[4];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][3] = 5;
Tr[TCurr][3] = 4;
}
/* PastDS 6, Transition 5 */
TotalDist = PathAccDist[6] + BranchDist[5];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][3] = 6;
Tr[TCurr][3] = 5;
}
/* PastDS 7, Transition 6 */
TotalDist = PathAccDist[7] + BranchDist[6];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][3] = 7;
Tr[TCurr][3] = 6;
}
NewPathAccDist[3] = PathAccDistMin;
if (NewPathAccDist[3] < GlobalDistMin)
{GlobalDistMin = NewPathAccDist[3];
BestDS = 3;}
/* NewDS 4 */
/* PastDS 0, Transition 3 */
PathAccDistMin = PathAccDis t[0] + BranchDist[3];
DS[TCurr][4] = 0;
Tr[TCurr][4] = 3;
/* PastDS 1, Transition 1 */
TotalDist = PathAccDist[1] + BranchDist[1];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][4] = 1;
Tr[TCurr][4] = 1;
}
/* PastDS 2, Transition 0 */
TotalDist = PathAccDist[2] + BranchDist[0];
if ( TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][4] = 2;
Tr[TCurr][4] = 0;
}
/* PastDS 3, Transition 2 */
TotalDist = PathAccDist[3] + BranchDist[ 2];
if (TotalDist < PathAccDistMin)
```

```
{
PathAccDistMin = TotalDist;
DS[TCurr][4] = 3;
Tr[TCurr][4] = 2;
}
NewPathAccDist[4] = PathAccDistMin;
if (NewPathAccDist[4] < GlobalDistMin)
{GlobalDistMin = NewPathAccDist[4];
BestDS = 4;
}
/* NewDS 5 */
/* PastDS 4, Transition 5 */
PathAccDistMin = PathAccDist[4] + BranchDist[5];
DS[TCurr][5] = 4;
Tr[TCurr][5] = 5;
/* PastDS 5, Transition 6 */
TotalDist = PathAccDist[5] + BranchDist[6];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][5] = 5;
Tr[TCurr][5] = 6;
}
/* PastDS 6, Transition 7 */
TotalDist = PathAccDist[6] + BranchDist[7];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][5] = 6;
Tr[TCurr][5] = 7;
}
/* PastDS 7, Transition 4 */
TotalDist = PathAccDist[7] + BranchDist[4];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][5] = 7;
Tr[TCurr][5] = 4;
}
NewPathAccDist[5] = PathAccDistMin;
if (NewPathAccDist[5] < GlobalDistMin)
{GlobalDistMin = NewPathAccDist[5];
BestDS = 5;
}
/* NewDS 6 */
/* PastDS 0, Transition 1 */
PathAccDistMin = PathAccDist[0] + BranchDist[1];
```

```
DS[TCurr][6] = 0;
Tr[TCurr][6] = 1;
/* PastDS 1, Transition 3 */
TotalDist = PathAccDist[1] + BranchDist[3];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][6] = 1;
Tr[TCurr][6] = 3;
}
/* PastDS 2, Transition 2 */
TotalDist = PathAccDist[2] + BranchDist[2];
if ( TotalDist < PathAccDistM in)
{
PathAccDistMin = TotalDist;
DS[TCurr][6] = 2;
Tr[TCurr][6] = 2;
}
/* PastDS 3, Transition 0 */
TotalDist = PathAccDist[3] + BranchDist[0];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][6] = 3;
Tr[TCurr][6] = 0;
}
NewPathAccDist[6] = PathAccDistMin;
if (NewPathAccDist[6] < GlobalDistMin)
{GlobalDistMin = NewPathAccDist[6];
BestDS = 6;
}
/* NewDS 7 */
/* PastDS 4, Transition 6 */
PathAccDistMin = PathAccDist[4] + BranchDist[6];
DS[TCurr][7] = 4;
Tr[TCurr][7] = 6;
/* PastDS 5, Transition 5 */
TotalDist = PathAccDist[5] + BranchDist[5];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][7] = 5;
Tr[TCurr][7] = 5;
}
/* PastDS 6, Transition 4 */
TotalDist = PathAccDist[6] + BranchDist[4];
if (TotalDist < PathAccDistMin)
```

```
{
PathAccDistMin = TotalDist;
DS[TCurr][7] = 6;
Tr[TCurr][7] = 4;
}
/* PastDS 7, Transition 7 */
TotalDist = PathAccDist[7] + BranchDist[7];
if (TotalDist < PathAccDistMin)
{
PathAccDistMin = TotalDist;
DS[TCurr][7] = 7;
Tr[TCurr][7] = 7;
}
NewPathAccDist[7] = PathAccDistMin;
if (NewPathAccDist[7] < GlobalDistMin)
{GlobalDistMin = NewPathAccDist[7];
BestDS = 7;
}
UpdateAccDistances(PathAccDist, NewPathAccDist);
return(BestDS);
}



//*******************************************************************
//   @Function   void   UpdateAccDistances   (int   PathAccDist[8],   int
NewPathAccDist[8])
//
//-------------------------------------------------------------------
// @DescriptionThis function updates the accumulated distance values by
storing
// them in the array PathAccDist.
//
//-------------------------------------------------------------------
// @Returnvaluenone
//
//-------------------------------------------------------------------
// @ParametersNewPathAccDist[8]: buffer where the accumulated distances
have
// been temporary stored during the current time
// period
//
// PathAccDist[8]: buffer where the new accumulated distances
// are transferred for each path
//
//-------------------------------------------------------------------
// @Date 19/02/99 15:00:00 AM
```

Appendix A - C Code Implementation for the Viterbi Algorithm

```
//
//*******************************************************************
void UpdateAccDistances (int PathAccDist[8], int NewPathAccDist[8])
{
#pragma asm
ld.d e14,[a5+]0x8
st.d [a4+]0x8,e14
ld.d e14,[a5+]0x8
st.d [a4+]0x8,e14
ld.d e14,[a5+]0x8
st.d [a4+]0x8,e14
ld.d e14,[a5+]0x8
st.d [a4+]0x8,e14
#pragma endasm
}



//*******************************************************************
// @Function _inline
// char TraceBack (char StartDS)
//
//-------------------------------------------------------------------
// @DescriptionThe purpose of this function is to determine the most
probalble
// transition by tracing back the maximum-likelihood path through
// the trellis once it has been identified and stored
//-------------------------------------------------------------------
//  @Returnvaluedifferential  encoded  data  Y1Y2Out,  obtained  after
discarding
// the MSB of the transition[Y0 Y1 Y2] selected as the most likely one
//
//-------------------------------------------------------------------
// @ParametersStartDS: start delay state to perform the trace back
operation
//
//-------------------------------------------------------------------
// @Date 19/02/99 15:00:00 AM
//
//*******************************************************************
_inline
char TraceBack (char StartDS)
{
char PastDS, T_TB, i;
T_TB = TCurr;
for (i = 0; i < LENGTH_TB-1; i++)
{
```

Appendix A - C Code Implementation for the Viterbi Algorithm

```
PastDS = DS[T_TB][StartDS];
StartDS = PastDS;
T_TB--;
if (T_TB < 0)T_TB = LENGTH_TB-1;
}
return( Tr[T_TB][StartDS]);
}



//******************************************************************
// @Function char ViterbiDecoding (char Y0Y1Y2_received)
//
//------------------------------------------------------------------
// @Descriptionknowing the transition received and using the structure
of the
// trellis (i.e. the allowed transitions), this function determines the
most
// likely path through the trellis for LENGTH_TB time periods and
// trace it back to find the most likely transmitted input symbol at
this time
// This means that the output for the current time reflects a decision
made
// by the decoder on data received up to LENGTH_TB time periods later.
//
//------------------------------------------------------------------
// @Returnvaluedifferential encoded data Y1Y2Out, obtained after
discarding
// the MSB of the transition[Y0 Y1 Y2] selected as the most likely one
// after a trace back of LENGTH_TB time periods
//
//------------------------------------------------------------------
// @ParametersY0Y1Y2_received: transition received by the decoder
//
//------------------------------------------------------------------
// @Date 19/02/99 15:00:00 AM
//
//******************************************************************
char ViterbiDecoding (char Y0Y1Y2_received)
{
static char BranchDist[8];
static int NewPathAccDist[8];
static char StartDS;
static char Y0Y1Y2selected, Y1Y2Out;
/* Compute the 8 current Hamming distance values between the input
symbol
and the 8 possible Transitions */
```

Appendix A - C Code Implementation for the Viterbi Algorithm

```
ComputeBranchDistances (Y0Y1Y2_received, BranchDist);
/* Metric Update */
StartDS = MetricUpdate (BranchDist, NewPathAccDist);
/* Select the most likely transition by tracing backward the most likely
path */
Y0Y1Y2selected = TraceBack (StartDS);
/* The MSB Y0 of the transition selected is discarded */
Y1Y2Out = Y0Y1Y2selected & 3;
return (Y1Y2Out);
}



//*********************************************************************
// @Function char DiffDecoding (char Y1Y2Out)
//
//---------------------------------------------------------------------
// @DescriptionThis function performs the differential decoding
// The encoded data Y1Y2Out is differentially decoded in an
// output symbol Q1Q2Out
//
//---------------------------------------------------------------------
// @Returnvaluedecoded output symbol Q1Q2Out
//
//---------------------------------------------------------------------
// @ParametersY1Y2Out: 2-bit differentially encoded data, output from
the
// convolutional decoder
//
//---------------------------------------------------------------------
// @Date 19/02/99 15:00:00 AM
//
//*********************************************************************
char DiffDecoding (char Y1Y2Out)
{
char Q1Q2Out;
Q1Q2Out = DiffDecod[PastY1Y2Out][Y1Y2Out];
PastY1Y2Out = Y1Y2Out;
return (Q1Q2Out);
}



//*********************************************************************
// @Function void IncrementTimePeriod (void)
//
//---------------------------------------------------------------------
```

Appendix A - C Code Implementation for the Viterbi Algorithm

```
//  @DescriptionOnce  all  steps  of  the  Viterbi  algorithm  have  been
performed, the
// current time period pointer Tcurr is incremented to enable a new pass
// of the Viterbi algorithm.
// Tcurr is necessary lower than LENGTH_TB since the metric storage
// buffers DS and Tr are set up as circular buffers.
//
//--------------------------------------------------------------------
// @Returnvaluenone
//
//--------------------------------------------------------------------
// @Parametersnone
//
//--------------------------------------------------------------------
// @Date 19/02/99 15:00:00 AM
//
//********************************************************************
void IncrementTimePeriod (void)
{
TCurr++;
if (TCurr >= LENGTH_TB)
TCurr = 0;
}
```

11 Appendix B - Benchmark

The breakpoints for benchmarking are placed before the call of the ViterbiDecoding function and after the call of the DiffDecoding function in the main function. Thus the complete decoding algorithm is considered.

Implemented Method	**Cycles**
Viterbi decoder + differential decoder	445

Figure 17 Clock cycle requirement for the V.32 Decoding Algorithm

Note: V.32 modems have a rate of 2400 symbols per second (baud).

Using this C implementation, at 2400 symbols/sec, the percentage loading of a 100Mhz computer is equal to:445*2400/100.10 6 = **1.07 Mips**

Infineon goes for Business Excellence

“Business excellence means intelligent approaches and clearly defined processes, which are both constantly under review and ultimately lead to good operating results.
Better operating results and business excellence mean less idleness and wastefulness for all of us, more professional success, more accurate information, a better overview and, thereby, less frustration and more satisfaction.”

Dr. Ulrich Schumacher

http://www.infineon.com

Published by Infineon Technologies AG