. .eescale Semiconductor
Application Note

AN2252
Rev. 3, 3/2005

Path Searcher for a WCDMA Rake Receiver

By Kim-Chyan Gan

Wideband CDMA (WCDMA), a third-generation air interface, is based on direct-sequence (DS) CDMA technology. A Rake receiver minimizes distortion of the signals in a DS-CDMA system. One component of a Rake receiver is a path searcher. The Rake receiver consists of “fingers,” each corresponding to one path delay of a received signal. The number of fingers and the delay of each finger in a Rake receiver are allocated on the basis of the path searcher. The operating parameters of the path searcher are assigned by the finger management.

In wireless channels, a signal may go through different fading paths. Since each path has a different length and the signals travel at the same speed, the signal arrival times for the paths differ. The path searcher finds the arrival time of the signals from different paths. In order for the path searcher to work, the delay of different paths must be greater than a chip period. Since the bandwidth of a CDMA system is usually wide, a chip period is very small, so delays of different paths are usually greater than one chip period.

This application note explains how the path searcher works and then describes how it is implemented on Freescale StarCore™-based DSPs.

CONTENTS

1 Received Model

In a DS-CDMA system, the discrete-time received signal from base station to handset can be expressed as follows:

Equation 1

$$r(p) = \sum_{k=1}^{K}\sum_{l=1}^{L} c_l(p)a_k\left(p - \left[\frac{\tau_l}{T_c}\right]\right) + n_{k,l}(p)$$

Where:

- p: discrete time index
- $r(p)$: received baseband signal after the match filter
- K: number of channels within frequency band
- L: number of fading paths
- $c_l(p)$: discrete complex channel coefficient of l-th path (channel distortion)
- $a_k(p)$: discrete transmitted chip data of k-th channel after scrambling and spreading
- τ_l: delay of l-th path
- T_c: chip period
- $[x]$: rounding of x
- $n_{k,l}(p)$: additive white gaussian noise (AWGN) of k-th channel in l-th path

The expression in **Equation 1** omits the symbol-level details and starts at the chip level. In a downlink, each channel within the same frequency band is separated by a spreading code. From the receiver point of view, every single channel within a frequency band goes through the same L-path fading. Thus, the complex channel coefficients are the same at a receiving point for all channels within a frequency band.

The path searcher is based on the Primary Common Pilot Channel (P-CPICH). In the WCDMA system, each frame contains 15 slots. In each slot of P-CPICH, ten complex symbols of 1+*j* are transmitted when there is one transmission source (no transmit diversity). Each symbol is expanded into chip level by a fixed spreading factor of 256 with a spreading code of "all ones." P-CPICH carries data, spreading code, and scrambling code. However, data is 1+*j* and the spreading code consists of 256 ones, so the only valuable information extracted from P-CPICH is the scrambling code. In **Equation 2**, the first term on R.H.S (*k*=1) refers to P-CPICH.

Equation 2

$$\begin{aligned} r(p) &= \sum_{l=1}^{L} c_l(p)sc\left(p - \left[\frac{\tau_l}{T_c}\right]\right) + \sum_{k=2}^{K}\sum_{l=1}^{L} c_l(p)a_k\left(p - \left[\frac{\tau_l}{T_c}\right]\right) + n_{k,l}(p) \\ &= \sum_{l=1}^{L} c_l(p)sc\left(p - \left[\frac{\tau_l}{T_c}\right]\right) + mai(p) + n_{k,l}(p) \\ &= \sum_{l=1}^{L} c_l(p)sc\left(p - \left[\frac{\tau_l}{T_c}\right]\right) + w(p) \end{aligned}$$

where

$$mai(p) = \sum_{k=2}^{K} \sum_{l=1}^{L} c_l(p) a_k\left(p - \left[\frac{\tau_l}{T_c}\right]\right)$$

mai(p) is the multiple-access interference caused by signals from other channels. *sc(p)* is the scrambling code for a particular cell. For simplicity, *mai(p)* can be modeled as AWGN. Both the *mai(p)* and $n_{k,l}(p)$ terms can be combined and become *w(p)*, which is also AWGN. There is some interference caused by signals transmitted from adjacent cells. However, inter-cell interference is assumed to be negligible.

2 Path Searcher

The path searcher has five blocks (see **Figure 1**). Initially, a received signal, *r(t)*, is digitized by A/D. Since the delay of each path may not be a multiple of the chip period, *r(t)* is over-sampled for better precision in a digitalization process. The discrete received signal is fed into the autocorrelation block, and the output of this block is the power delay profile based on one frame of P-CPICH. In the next block, the power delay profile is averaged over a few previous frames. The finger management algorithm sets the tap, length, and frame parameters, which determine how the complex autocorrelation and average block are performed. Next, a search for a local peak is performed on the averaged power delay profile while a threshold is computed using just the first moment statistics of the averaged power delay profile. Finally, the algorithm determines the validity of strong peaks by checking their magnitude against the threshold. A strong peak corresponds to one path signal. The index of the strong peaks, which is delay of signals in discrete form, is passed from the path searcher to the Rake receiver.

2.1 Complex Autocorrelation

Complex autocorrelation is the most computationally intensive part of the path searcher. Initially, the cell search gives an estimate of when the frame starts, and the path searcher searches around this estimated starting point. The path searcher relies on the near orthogonal[1] property of the complex scrambling code that separates each path signal. The multi-path fading signal in P-CPICH can be viewed as a multiple shifted version of the scrambling code combined[2]. The autocorrelation yields to a peak whenever there is an exact alignment between the conjugate of the scrambling code (*sc* (t)*) and the delay/shift version of the scrambling code embedded in P-CPICH (*r(t)*). Thus, three peaks imply three strong path signals.

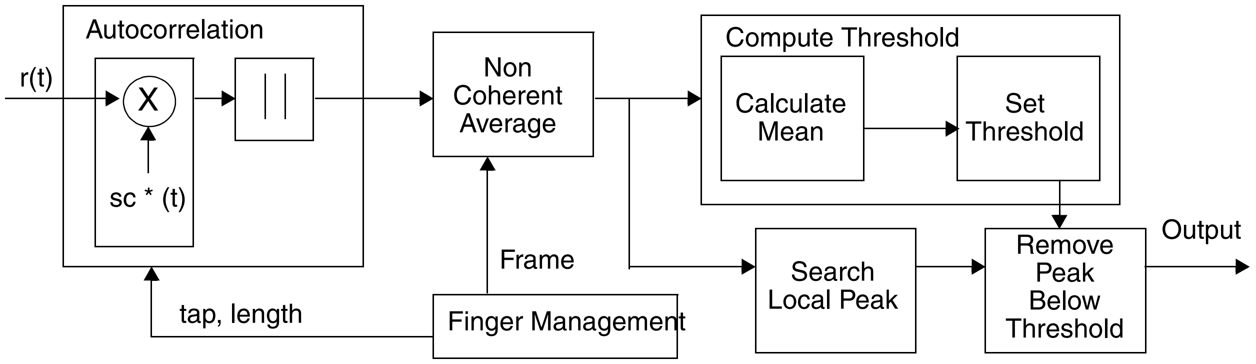

Figure 1. Block Diagram of Path Searcher

1. Two vectors, a and b are orthogonal if <a,b> is equal to zeros, where < > is the inner vector product.
2. Each channel consists of data, spreading code, and scrambling code. P-CPICH bears only scrambling code since data is *1+j* and the spreading code consists of 256 ones. In a multi-path channel, P-CPICH has different paths, and each path has it own delay. The composite received signal can be viewed as a superposition of the multiple-delay version of the scrambling code.

The received signal is correlated with the conjugate of the scrambling code, which is unique within the cell in which the base station is located. The output of the correlator can be expressed as follows:

Equation 3

$$\begin{aligned} y(n,\ m) &= \frac{1}{N}\sum_{p=0}^{N-1} r(p+m)sc^*(p),\ m = 0,\ 1,\ 2..P-1 \\ &= \frac{1}{N}\sum_{p=0}^{N-1}\sum_{l=1}^{L} c_l(p+m)sc\left(p+m-\left[\frac{\tau_l}{T_c}\right]\right)sc^*(p) + w(p+m)sc^*(p),\ m = 0,\ 1,\ 2..P-1 \end{aligned}$$

where N is the “no of taps” or the “window size” of the autocorrelation chosen, P is the “length” of autocorrelation, m is the “index” of autocorrelation, p is the discrete time index, and n is the frame index. For the path searcher to tolerate up to 20 µs of delay spread, P is chosen to be 320. If $m=[\tau_i/T_c]$ when the autocorrelation index is equal to delay of one of the paths, the $y(n,m)$ becomes:

Equation 4

$$y(n,\ m) = \frac{1}{N}\left(\sum_{p=0}^{N-1} c_i(p)\delta\left(m-\left[\frac{\tau_i}{T_c}\right]\right) + \sum_{p=0}^{N-1}\sum_{l=1,\ l\neq i}^{L} c_l(p)sc\left(p+m-\left[\frac{\tau_l}{T_c}\right]\right) + \sum_{p=0}^{N-1} w(p+m)sc(p)\right)$$

The first term on the R.H.S. is the average of the channel coefficients, which is a complex gaussian distribution. The summation and scaling of the first term on the R.H.S. is essentially the same as taking the mean of the channel coefficient. Since N is not large enough, the first term on the R.H.S. becomes the local mean of the channel coefficient. The second term on the R.H.S. is interpath interference (IPI), which is interference caused by other paths and does not align perfectly with the scrambling code. Since the scrambling code yields a peak only if two scrambling codes are perfectly aligned, the IPI is usually small when compared with the third term, which is multi-access interference (MAI) and thermal noise. Usually, the first term is larger than the last two terms. However, in deep fading duration when the local mean of channel coefficients is close to zero, the magnitude of the first term can be significantly lower, undermining the performance of path searcher.

If m does not align with any of the path delays, the autocorrelation output becomes the following, which consists of only interference and noise terms.Typically, these two terms are small.

Equation 5

$$y(n,\ m) = \frac{1}{N}\left(\sum_{p=0}^{N-1}\sum_{l=1}^{L} c_l(p)sc\left(p+m-\left[\frac{\tau_l}{T_c}\right]\right) + \sum_{p=0}^{N-1} w(p+m)sc(p)\right)$$

2.2 Non-Coherent Averaging

Non-coherent averaging helps to combat a spurious peak signal in a fading channel. It averages the current and previous $M-1$ power delay profiles since a single measurement of a power delay profile is likely to be erroneous. M can be chosen to be a power of two so that the division is implemented efficiently by scaling. Non-coherent averaging can be expressed as follows:

Equation 6

$$z(n,\ m) = \frac{1}{M}\sum_{k=0}^{M-1} |y(n-k,\ m)|^2,\ m = 0,\ 1,\ 2..P-1$$

where *M* is the total number of power delay profiles to be averaged, *P* is the "length" of the autocorrelation, *m* is the autocorrelation index, and *n* is the frame index. Averaging the power delay profile reduces the spurious floor noise, especially when there is high interference. If the channel fluctuates slowly, *M* should be increased to get a more accurate estimate. If the power delay profile changes rapidly, *M* should be decreased to increase the adaptive rate for a better performance. This process is characterized as "non-coherent" since averaging elements are obtained from different time intervals (from frame index *n-M-1* to frame index *n*). Assuming that the channel statistic does not change over *M* frame, **Equation 6** can be simplified, as follows:

Equation 7

$$z(n,\ m) = |E\{c_i(p)\}|^2\delta\left(m - \left[\frac{\tau_i}{T_c}\right]\right) + \sigma^2,\ m = 0,\ 1,\ 2..P-1$$

The first term contributes to the peak in the power delay profile, and the second term constitutes the floor noise. If the autocorrelation index, *m*, does not correspond to the delay, the first term disappears, leaving only the second term.

2.3 Local Peak Search

The local peak search searches for all the local peaks in the power delay profile. It is based on the observation of three points. As long as the middle point is higher than the two points at the side, a local peak is found. Both the magnitude and index of the peaks are stored.

2.4 Compute Threshold

Not all local peaks correspond to the delay index. A threshold is computed, which should be higher than all the floor noise but lower than the true delay peak. Usually, the component of floor noise comes from interpath interference (IPI), multi access interference (MAI), and thermal noise. An adaptive threshold is used because it is more robust, accounting for both interference and noise variations. The threshold is based on statistics of the power delay profile. Since all the terms in **Equation 7** are complex gaussian distributions, the first and second moments are enough to determine the statistics of the power delay profile. In [1], the mean and variance of the power delay profile are calculated for a threshold setting. A simplified way of threshold setting based only on the mean is presented in [2]. The threshold is given in **Equation 8**.

Equation 8

$$\theta = \frac{1}{P}\left(\sum_{m=0}^{P-1} z(n,\ m)\right)(a + bM^c)$$

where *M* is number of frames to be used in calculating the power delay profile, and *a*,*b*, and *c* are the parameters that can be adjusted for good performance. In [2], experiments are conducted to find the different value of *a*, *b*, and *c* for better performance. *a*=2, *b*=4, *c*=-0.5 yields the best results. The formula used in the implementation is:

Equation 9

$$\theta = \frac{1}{P}\left(\sum_{m=0}^{P-1} z(n, m)\right)\left(2 + \frac{4}{\sqrt{M}}\right)$$

2.5 Local Peak Removal

Local peak removal is the final operation of the path searcher. All the local peaks are compared against the threshold, and peaks lower than the threshold are removed and the higher are retained. In this step, the noise-only path is discarded because it would penalize the decoding performance. It also increases the power consumption by dedicating a redundant finger to do the computation. However, if the number of paths, denoted by L, is greater than number of fingers available in the Rake receiver, denoted by N, the finger management assigns the highest N peaks of the power delay profile to the N finger.

3 Implementation in StarCore

The main path searcher routine is written in "C," since the code for this routine is mostly control-oriented. The implementation is based on **Figure 1**. The pseudo-code of this routine is as follows:

1. Perform complex autocorrelation. In this implementation, there are two types of autocorrelation, exhaustive and fast. The output of the autocorrelation is the power delay profile of the received signal:
   - Exhaustive autocorrelation is performed in the oversampling chip period (which is a factor of four smaller than the chip period) over 20 µs. It is performed once every *Mth* frame (*mod_idx* is equal to zero).
   - Fast autocorrelation is performed in the chip period (which is a factor of four larger than the over-sampling chip period in this example). It is then interpolated by a factor of four using a zero-order hold approach (that is, simply repeated) over 20 µs. Then autocorrelation is performed in the oversampling chip period around the peak of the previous averaged power delay profile. Fast autocorrelation is performed if *mod_idx* is not equal to zero.
2. Average the power delay profile over M frames.
3. Find the local peaks and the threshold on the basis of the averaged power delay profile.
4. Remove the peaks below the threshold.

Example 1. Path Searcher

```
#include <stdlib.h>
#define AVG_LEN 8        /* M in Eqn 6 */
#define AUTO_LEN 320     /* P in Eqn 6 */
static long autoc[AVG_LEN+1][AUTO_LEN];
#pragma align *autoc 16
static short mod_idx=0;
int pathsearch(short *in, short *sc, short *peak_idx){
#pragma align *sc 8
#pragma align *in 8
#pragma align *peak_idx 8
static short prev_idx[300];
  long mag[300];
#pragma align *mag 16
  short idx[300];
#pragma align *idx 16
  short no_peak;
```

```
    long threshold;
    short *tmp_ptr;
    short i;
    if (mod_idx == 0){
      FullComplexAuto(&in[0],sc,&mag[0]);
    }
    else {
      ComplexAuto(&in[2],sc,mag);
      for (i=1; i<=prev_idx[0] ; i++){
        FineComplexAuto(&in[(prev_idx[i]-4)*2],sc,&mag[prev_idx[i]-4]);
      }
    }
    average(mag);
    no_peak = findlocalpeak(&mag[0], &idx[0]);
    threshold = findthreshold();
    no_peak = removepeak(mag, idx, threshold, no_peak);
    prev_idx[0] = no_peak;
    for (i=0; i<no_peak; i++){
      prev_idx[1+i] = idx[i];
      peak_idx[i] = idx[i];
    }
    mod_idx = ((mod_idx+1) % AVG_LEN);
    return (no_peak);
  }
```

3.1 Complex Autocorrelation

The received signal contains an enormous amount of data. In every *M* frames, exhaustive complex autocorrelation in the oversampling period is performed. For the rest of the frames, fast complex autocorrelation is performed in the chip period and interpolated to the oversampling chip period. Then, autocorrelation is also performed in every oversampling chip interval around the peaks of the previous averaged power delay profile because the autocorrelation peaks probably occur around the same index with a slight variance. This approach decreases the computation without significantly sacrificing performance. The length of autocorrelation chosen is 320, which allows the system to tolerate a delay spread of up to 20 µs. This interval is larger than the typical delay of an urban (3 µs) and sub-urban (0.5 µs) environment [4]. The output of a complex autocorrelation is a power delay profile, which is a series of complex numbers (see **Equation 4**). Different metrics, for example L1 or other metrics, can be used to evaluate the “pseudo” power delay profile for simplicity of computation. If the *L2* metric is taken, *y(n,m)* becomes the power delay profile. The metric implemented in this example is the absolute value of the real plus the absolute value of the imaginary. The optimization techniques employed in this subroutine are as follows:

- *Guard bit*. There is a total of 8 guard bits in data registers (d0 to d15). These bits ensure 256 additions without overflowing the register. The scaling occurs after 256 additions. However, to use the guard bit, the saturation mode in the SR register must be turned off at the beginning of the routine and on at the end of the routine.
- *Software pipelining*. Ensures that the DALUs are fully employed in the inner loop. The data is loaded into the data register before the loop. The memory-to-register load in the last execution set of the “inner3” loop is used in the next loop.
- *Multisampling*. Prevents memory alignment problems and exploits the inter-tap data redundancy, decreasing memory traffic [5]. However, two autocorrelations must be computed within the “inner3” loops, implying that the length of autocorrelation must be multiple of two.

Example 2 uses the exhaustive autocorrelation. Two functions in a fast autocorrelation implementation are ComplexAuto() and FineComplexAuto() (see **Example 1**). Since they are similar to the implementation in **Example 2**, their source code is not included. The code shown in **Example 2** is written in stand-alone assembly and follows the application binary interface (ABI). The complex autocorrelation is stored in 32 bits instead of 16 bits to accommodate a large dynamic range. This routine requires 839,231 cycles and has a code size of 242 bytes.

Example 2. Complex AutoCorrelation

```
section .text local
    global  _FullComplexAuto
    align       16
_FullComplexAuto    type func
; r0           : in -- r6
; r1           : sc  -- r5
; (sp-28): idx -- r3
     [ push r6
       push r7
     ]
     [ push d6
       push d7
     ]
     [ bmclr #$4,sr.l                ; saturation off
       tfra r0,r6                    ; r6 -> in
     ]
     [ move.l (sp-28),r3             ; r3 -> idx
       move.l #4,n0
     ]
       adda #4,r6,r7
     [ dosetup1 inner1
       doen1 #160                   ; AUTO_LEN/2
     ]
      loopstart1
inner1
     [ clr d12
       clr d13
       clr d14
       clr d15
       tfra r1,r5                    ; r5 -> sc
       move.2f (r6)+n0,d4:d5         ; load in_r(1),in_i(1)
     ]
     [ move.2f (r5)+,d0:d1           ; load sc_r(1),sc_ri(1)
       move.2f (r7)+n0,d6:d7         ; load in_r(4),in_i(4)
     ]                               ; software pipelining
     [ dosetup2 inner2
       doen2 #10
     ]
     loopstart2
inner2
     [ clr d8
       clr d9
       clr d10
       clr d11
     ]
     [ dosetup3 inner3
       doen3 #128
     ]
      loopstart3
inner3
     [ mac d4,d0,d8                  ; A_r(1) += in_r(1) * sc_r(1)
```

```
  mac d5,d0,d9                ; A_i(1) += in_i(1) * sc_r(1)
  mac d6,d0,d10               ; A_r(4) += in_r(4) * sc_r(1)
  mac d7,d0,d11               ; A_i(4) += in_i(4) * sc_r(1)
  move.2f (r6)+n0,d2:d3       ; load in_r(4),in_i(4)
]
[ mac  d5,d1,d8               ; A_r(1) +=  in_i(1) * sc_i(1)
  mac -d4,d1,d9               ; A_i(1) += -in_r(1) * sc_i(1)
  mac  d7,d1,d10              ; A_r(4) +=  in_i(4) * sc_i(1)
  mac -d6,d1,d11              ; A_i(4) += -in_r(4) * sc_i(1)
  move.2f (r5)+,d0:d1         ; load sc_r(2),sc_ri(2)
  move.2f (r7)+n0,d6:d7       ; load in_r(8),in_i(8)
]
[ mac d2,d0,d8                ; A_r(1) += in_r(4) * sc_r(2)
  mac d3,d0,d9                ; A_i(1) += in_i(4) * sc_r(2)
  mac d6,d0,d10               ; A_r(4) += in_r(8) * sc_r(2)
  mac d7,d0,d11               ; A_i(4) += in_i(8) * sc_r(2)
  move.2f (r6)+n0,d4:d5       ; load in_r(8),in_i(8)
]
[ mac  d3,d1,d8               ; A_r(1) +=  in_i(4) * sc_i(2)
  mac -d2,d1,d9               ; A_i(1) += -in_r(4) * sc_i(2)
  mac  d7,d1,d10              ; A_r(4) +=  in_i(8) * sc_i(2)
  mac -d6,d1,d11              ; A_i(4) += -in_r(8) * sc_i(2)
  move.2f (r5)+,d0:d1         ; load sc_r(3),sc_ri(3)
  move.2f (r7)+n0,d6:d7       ; load in_r(12),in_i(12)
]
 loopend3
[ asrr #14,d8                 ; scaling
  asrr #14,d9
  asrr #14,d10
  asrr #14,d11
]
[ add d8,d12,d12              ; A_r(1)
  add d9,d13,d13              ; A_i(1)
  add d10,d14,d14             ; A_r(4)
  add d11,d15,d15             ; A_i(4)
]
 loopend2
[ abs d12
  abs d13
  abs d14
  abs d15
  adda #8,r0,r0
]
[ add d12,d13,d12             ; |A_r(1)| + |A_i(1)|
  add d14,d15,d13             ; |A_r(4)| + |A_i(4)|
  tfra r0,r6                  ; r6 -> in
  tfra r1,r5                  ; r5 -> sc
]
[ move.2l d12:d13,(r3)+
  adda #4,r6,r7
]
 loopend1

  bmset #$4,sr.l              ; saturation mode on
  nop
  nop
[ pop d6
  pop d7
]
[ pop r6
```

```
        pop r7
      ]
        rts
      endsec
```

3.2 Non-Coherent Averaging

Non-coherent averaging is based on **Equation 6** and assumes that the length of the autocorrelation is *P* and the number of the frame to be averaged is *M*. The total number of additions required is $(M-1)\times P$. However, the number of additions can be reduced to $2\times P$ at the expense of memory. The savings is large when *M* increases. The approach is based on circular first-in-first-out (FIFO) memory, as shown in
Figure 2.

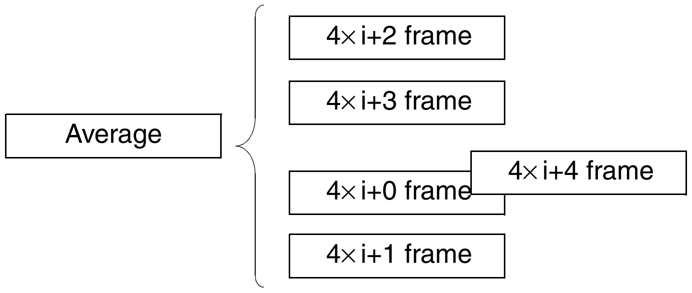

Figure 2. Averaging

Assume that four frames are averaged and that the average buffer contains an average of frame 4i+0, 4i+1, 4i+2, and 4i+3. As the new frame (4× i+4) arrives, the average is recomputed by subtracting the value in the older frame (4× i+0) and adding up the value of the latest frame (4× i+4). Now, the average buffer contains the average value of the 4i+1, 4i+2, 4i+3, and 4i+4 frames. The routine shown in **Example 3** is written in in-line assembly format with a C interface. A “%C” is appended to the label so that compiler does not issue “duplicate label” error message when the optimization option is turned on. This routine requires 505 cycles and has a code size of 112 bytes.

Example 3. Non-Coherent Averaging

```
#if NOOPT
void average(long *idx){
  unsigned int i,j;
  for (i=0 ; i<AUTO_LEN ; i++){
    autoc[AVG_LEN][i] += idx[i];
    autoc[AVG_LEN][i] -= autoc[mod_idx][i];
    autoc[mod_idx][i] = idx[i];
  }
}
#else
asm void average(long *idx)
{
asm_header
.arg
  _idx in $r0;
.reg $d0,$d1,$d2,$d4,$r0,$r1,$r2,$r3;
asm_body
      [ push d6
        push d7
      ]
        bmclr #$4,sr.l
        move.l #_autoc,d0
        move.l #_mod_idx,r1
      [ move.w #1280,d7
        move.w #160,d1                ; AUTO_LEN = 320/2
      ]
        move.w (r1),d2
      [ imac d7,d2,d0
        move.l #_autoc+10240,r2
      ]
      [ move.l d0,r1
        doen0 d1
      ]
      [ dosetup0 _lp0%C
        tfra r2,r3
      ]
      [ move.2l (r0)+,d0:d1           ; idx[i]
        move.2l (r3)+,d8:d9           ; autoc[AVG_LEN][i]
      ]
        falign
       loopstart0
_lp0%C:
      [ move.2l d0:d1,(r1)+           ; autoc[mod_idx][i] = idx[i]
        move.2l (r1),d2:d3            ; autoc[mod_idx][i]
        add d8,d0,d4                  ; autoc[AVG_LEN][i] + idx[i]
        add d9,d1,d5
      ]
      [ sub d2,d4,d4                  ; autoc[AVG_LEN][i] - autoc[mod][i]
        sub d3,d5,d5
        move.2l (r0)+,d0:d1           ; idx[i]
        move.2l (r3)+,d8:d9           ; autoc[AVG_LEN][i]
      ]
        move.2l d4:d5,(r2)+
       loopend0
        bmset #$4,sr.l
        nop
        nop
      [ pop d6
        pop d7
      ]
asm_end
}
#endif
```

3.3 Find Local Peak

The local peak search is based on three consecutive points, *a*, *b*, and *c*. The gradient of *a* and *b* is calculated in the previous loop and stored in an “up” buffer. The gradient between *b* and *c* is calculated in the current loop. Based on these gradients, the routine determines whether a local peak is found. A local peak is found only if the *a*-*b* gradient is positive and the *b*-*c* gradient is negative. Since it is a sliding window, the *b*-*c* gradient become the *a*-*b* gradient in the next iteration. Thus, the *b*-*c* gradient is stored in the “up” buffer. The code shown in **Example 4** is written in in-line assembly format with a C interface. It requires 2078 cycles and has a code size of 90 bytes. It is control-oriented and therefore contains checking and branching, which break the pipeline. This routine searches through all the local peaks and returns the number of each local peak, its value, and its index.

Example 4. Find Local Peak

```
#if NOOPT
short findlocalpeak(long *mag, short *idx){
  short up=0,ii=0;
```

```
    unsigned short i;
    for (i=1 ; i< AUTO_LEN ; i++) {
      if (autoc[AVG_LEN][i] - autoc[AVG_LEN][i-1] < 0){
        if (up == 1) {
         idx[ii] = i-1;
         mag[ii] = autoc[AVG_LEN][i-1];
         ii++;
        }
        up = 0;
      }
      else {
        up = 1;
      }
    }
    return (ii); // number of local peak
  }
  #else
  asm short findlocalpeak(long *mag, short *idx)
  {
  asm_header
  .arg
   _mag in $r0;
   _idx in $r1;
   return in $d0;
  .reg $d0,$d1,$d2,$d4,$d5,$d8,$d9,$r0,$r1,$r2;
  asm_body
         move.l #_autoc+10240,r2
       [ clr d4                         ; up = d4
         clr d3                         ; ii = d3
         move.w #-1,d2                  ; i = d2
       ]
         move.2l (r2)+,d0:d1
       [ doen3 #319                     ; AUTO_LEN
         dosetup3 lllp3%C
       ]
  lllp3%C:
         falign
        loopstart3
       [ sub d1,d0,d5                   ; autoc[AVG_LEN][i-1] - autoc[AVG_LEN][i]
         inc d2                         ; i = i+1
         tfr d0,d8                      ; prev
         tfr d1,d0                      ; prev = curr
         move.l (r2)+,d1
       ]
       [ tstgt d5
         tfr d4,d9                      ; up - d9
         clr d4
       ]
         nop
       [ iff cont end_lllp3%C
         move.w #1,d4
       ]
         cmpeq.w #1,d9
         nop
       [ ift inc d3
         move.w d2,(r1)+
         move.l d8,(r0)+
       ]
        loopend3
  end_lllp3%C:
         tfr d3,d0                     ; return (ii)
  asm_end
  }
  #endif
```

3.4 Compute Threshold

The computer threshold routine is based on **Equation 9** and is written in in-line assembly format with a C interface. It requires 106 cycles and has a code size of 120 bytes. Initially, the mean of the power delay profile is calculated. The threshold is set according to the mean of the power delay profile. In computing the mean, the additions are performed in a tree-like structure. One part of the values is added into register A and another part is added into register B. Eventually, register A and register B are added together.

Example 5. Compute Threshold

```
#if NOOPT
long findthreshold()
{
  unsigned short i;
  long m=0;
  for (i=0; i<AUTO_LEN ; i++){
    m += autoc[AVG_LEN][i] >> 8;
  }
  m = m*3; // aproximate mean * (2+4/sqrt(8))
  // 2.73 * m
  return(m);
}
#else
asm long findthreshold()
{
asm_header
 return in $d0;
.reg $d0,$d1,$d2,$d4,$d5,$d6,$d7,$r0,$r1,$r2,$r3;
asm_body
       bmclr #4,sr.l                 ; saturation off, use ext bits
       move.l #_autoc+10240,r0
       nop
     [ push d6
       push d7
     ]
     [ adda #8,r0,r1
       move.w #2,n0
     ]
     [ doensh0 #79
       dosetup0 _llp0%C
     ]
     [ move.2l (r0)+n0,d0:d1
       move.2l (r1)+n0,d2:d3
       clr d6
       clr d7
     ]
     [ add d0,d1,d4
       add d2,d3,d5
       move.2l (r0)+n0,d0:d1
       move.2l (r1)+n0,d2:d3
     ]
       falign
      loopstart0
_llp0%C:
     [ move.2l  (r0)+n0,d0:d1
       move.2l  (r1)+n0,d2:d3
       add d0,d1,d4
       add d2,d3,d5
       add d4,d6,d6
       add d5,d7,d7
     ]
      loopend0
     [ add d4,d6,d6
       add d5,d7,d7
     ]
       add d6,d7,d0

     [ asrr #8,d0
       move.w #3,d1
     ]
     [ impyuu d0,d1,d2
       impysu d0,d1,d3
     ]
       aslw d3,d3
       add d2,d3,d0
       bmset #$4,sr.l
       nop
       nop
     [ pop d6
       pop d7
     ]
asm_end
}
#endif
```

3.5 Local Peak Removal

The local peak removal routine checks all the local peaks against the threshold and eliminates the local peaks below the threshold. This routine is written in in-line assembly format with a C interface. It requires 2608 cycles and has a code size of 78 bytes.

Example 6. Local Peak Whacking

```
#if NOOPT
short removepeak(long *mag, short *idx, long threshold, short no){
  unsigned short i,ii=0;
  for (i=0 ; i<no ; i++){
    if (mag[i] > threshold){
      mag[ii] = mag[i];
      idx[ii] = idx [i];
      ii++;
    }
  }
  return(ii);
}
#else
asm short removepeak(long *mag, short *idx, long threshold, short no)
{
asm_header
.arg
 _mag in $r0;
 _idx in $r1;
 _threshold in $d5;
 _no in $r4;
 return in $d0;
.reg $d0,$d1,$d2,$d4,$d5,$r0,$r1,$r2,$r3;
asm_body
     [ tfra r0,r2                ; mag
       tfra r1,r3                ; idx
       clr d4                    ; ii = 0
     ]
here%C:
     [ doen0 r4                  ; loop no
       dosetup0 lp0%C
     ]
     [ move.l (r0)+,d0           ; mag[i] - d0
       move.w (r1)+,d1           ; idx[i] - d1
     ]
      loopstart0
lp0%C:
       cmpgt d5,d0               ; mag[i] > threshold
     [ move.l (r0)+,d0           ; mag[i] - d0
       move.w (r1)+,d1           ; idx[i] - d1
       tfr d0,d2
       tfr d1,d3
     ]
     [ ift inc d4                ; ii++
       move.l d2,(r2)+           ; mag[ii]
       move.w d3,(r3)+           ; idx[ii]
     ]
      loopend0
       tfr d4,d0
asm_end
}
#endif
```

4 Results

A total of 16 frames of data generated at 3dB SNR without multiple access interference (MAI) was used to test the routine. The length of autocorrelation, *P*, is set to 320, the frame to be average, *M*, is set to 8, and number of taps (window size) of autocorrelation, *N*, is set to 2560. **Table 1** shows the code size and approximate cycle per frame of each function of the path searcher. The code size for the whole path searcher routine is 1688 bytes. **Figure 3** shows the power delay profile in the sixteenth frame.

Table 1. Code Size and Approximate Cycle Count

Function	Approximate Cycles/Frame	Code Size (Bytes)
Full complex autocorrelation	839231	242
Fast complex autocorrelation	260703	256
Fine complex autocorrelation	20992	264
Non-coherent average	505	112
Local peak search	2078	90
Compute threshold	106	120
Local peak removal	2608	78

Table 2 shows the overall number of MIPS required by full and fast path searcher.

Table 2. MIPS Required For Path Searcher Algorithm

Path Searcher	Approximate Cycles/Frame	MIPS
Full search	844796	84.4
Fast search	328211	32.8

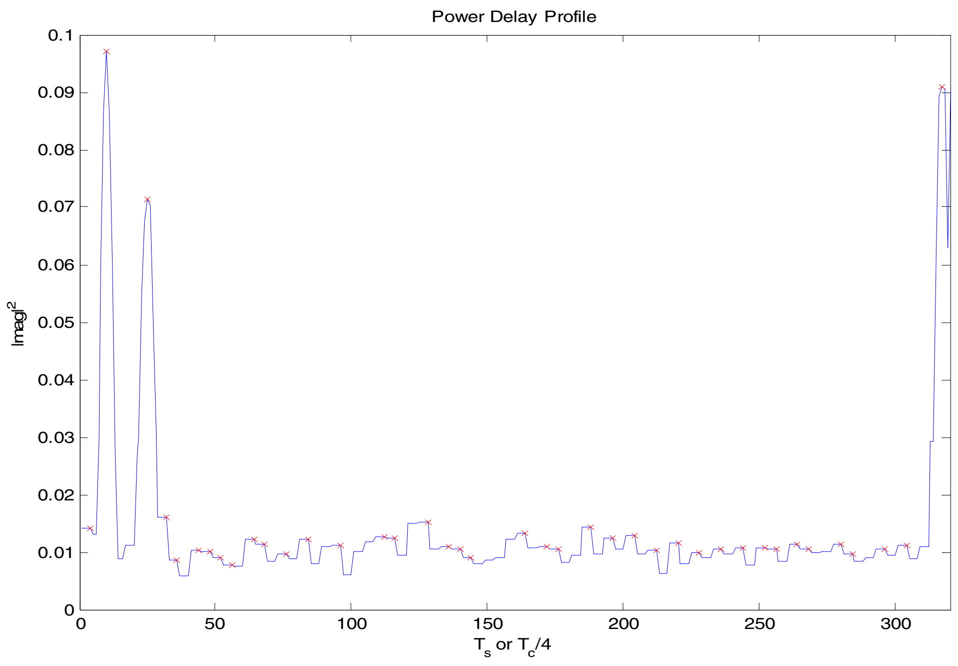

Figure 3. Power Delay Profile in the Sixteenth Frame

In the channel simulation, the path delays are set to [10 25 317] for the three paths. **Table 3** shows the delay index from the SC140 core.

Table 3. Peak Return from the Path Searcher Algorithm

Frame	Delay Index
1	10,318
2	10,317
3	10,25,28,317
4	10,25,317
5	10,25,317
6	10,25,317
7	10,25,317
8	10,25,317
9	10,25,317
10	10,25,317
11	10,25,317
12	10,25,317

Table 3. Peak Return from the Path Searcher Algorithm (Continued)

Frame	Delay Index
13	10,25,317
14	10,25,317
15	10,25,317
16	10,25,317

5 Fine Tuning

Some parameters in the path searcher can be adjusted in a trade-off between complexity and performance. Such adjustments allow deployment of wireless systems in different channel scenarios. These parameters are as follows:

- *Length of the autocorrelation* (*P*). Adjustment of the length mainly depends on the power delay profile of a particular channel scenario. The distribution of delays of received signal can be approximated by the exponential function. The earlier the signal comes in, the stronger the signal is. The delay also spread depends on the environment type. For example, delay spread in an urban area is larger than that in a suburban area. The AUTO_LEN, which controls the length of the autocorrelation in the code, should be adjusted depending on the expected delay spread. The AUTO_LEN set in this path searcher is 320 taps, which allow path searcher to catch a delay up to 20 µs.
- *Tap (window size) of the autocorrelation (N)*. Can be adjusted in the step size of 256. The larger the number of tap of the autocorrelation, the bigger the processing gain and the more robust it is in eliminating interference. However, it increases the MIPS linearly. It can be used to adjust the system load by increasing and decreasing the length.
- *Number of "frames" in the power delay profile to be averaged (M)*. The frame is used to adjust the average of the power delay profile. It depends on how fast a path changes in the environment. If the position of path stays almost the same for a long time, the "frame" should be increased so that it rejects the spurious peak. If a mobile unit moves quickly from one place to another, the delay of each path would probably change. In this case, the frame should be short so that its adaptive rate is faster.

Usually, finger management controls these parameters in the path searcher for good decoding performance, on the basis of an heuristic approach or statistics collected in real time.

6 References

[1] E. Bejjani, J.-F. Bouquier, and B. de Cacqueray. "Adaptive Channel Delay Selection for WCDMA Mobile System." *In Proceedings of IEEE VTC 1999*, pp. 203-207, Amsterdam, Netherlands, 1999.

[2] H. Elders-Boll. "Simplified Interference-Based Threshold Rule for Delay Selection in DS-CDMA Systems." PPIMRC 2000. *The 11th IEEE International Symposium*, Volume 1, pp. 77–81, 2000.

[3] J. G. Proakis. *Digital Communications*, McGraw-Hill, 4th edition, 2000.

[4] Lee, W. C. Y. *Mobile Communications Design Fundamentals*, 2nd ed. New York: Wiley, 1993.

[5] *StarCore Multisample Programming Technique Application Note* (STCR140MLAN/D).

NOTES:

NOTES:

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

AN2252
Rev. 3
3/2005