Application Note

07 April 2004

www.picochip.com

WiMax: OFDM Synchronization on the picoArray

Summary

The 802.16a-2003 standard specifies three different physical (PHY) layers, a host of possible configurations (e.g. TDD/FDD, channel bandwidth) and a number of optional features. The flexibility in the standard, limited availability of CPE equipment and time-to-market pressure make a software solution highly desirable, especially for the BS. However, the sampling rates involved make a full baseband implementation on single-processor DSPs untenable.

The picoArray^{TM} combines the programmability of a traditional high-end DSP with the performance of a FPGA/ASIC. The picoArray^{TM} is ideally suited for implementing the full baseband for the 802.16 suite of protocols. This application note looks in detail at an implementation of the receiver synchronization block for the WirelessMAN-OFDM uplink.

Table of Contents

Table of Figures

Abbreviations

AE	Array Element
ASIC	Application Specific Integrated Circuit
AWGN	Additive White Gaussian Noise
BS	Basestation
BWA	Broadband Wireless Access
CP	Cyclic Prefix
CPE	Customer Premises Equipment
DL	Downlink
DSP	Digital Signal Processor
FDD	Frequency Division Duplex
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
LIW	Long Instruction Word
NLOS	Non Line Of Sight
OFDM	Orthogonal Frequency Division Multiplexing
PDU	Protocol Data Unit
PHY	Physical layer
PMP	Point to Multipoint
SS	Subscriber Station
TDD	Time Division Duplex
UL	Uplink

1 Introduction

The picoArray™ is a multi-processor IC which integrates hundreds of processing elements into a single array. The individual elements have been optimized for signal processing and wireless algorithm computation and control. The result is a general purpose wireless communications processor, capable of executing all contemporary wireless standards, which combines the computational density of a dedicated ASIC with the programmability of a traditional high-end Digital Signal Processor (DSP).

WiMax is the industry forum name associated with the 802.16 suite of standards for broadband wireless access (BWA). WiMax offers raw bit rates in excess of 70Mbps (on a 20MHz channel), quality of service, authentication and data encryption over a metropolitan area.

Much of the focus is on the standard (currently 802.16a-2003) defined for NLOS operation in the licensed and unlicensed bands in the 2-11GHz range. Of particular interest are the physical (PHY) layers using orthogonal frequency division multiplexing (OFDM) given their simplicity and robustness in the presence of multipath.

The performance and flexibility offered by the picoArray^{TM} make it the ideal platform for implementing the plethora of options associated with a full WiMax BS baseband solution. This application note looks in detail at a picoArray^{TM} implementation of the receiver synchronization block for the WirelessMAN-OFDM PHY layer.

An overview of the receiver synchronization requirements and preamble used in the WirelessMAN-OFDM uplink is given in section 2. The key features of the PC102 picoArray^{TM} are described briefly in section 3 to assist understanding. A picoArray^{TM} implementation for obtaining coarse symbol timing is then given in section 4. This is complemented by a full source listing in section 5.

NOTE: The implementation presented in this application note is for illustrative purposes only. No claim is made with regard to conformance with the relevant standards.

2 WirelessMAN-OFDM

The WirelessMAN-OFDM PHY is one of the three physical layers defined in the 802.16a-2003 for NLOS operation in the 2-11GHz range. Each OFDM symbol consists of 256 sub-carriers. A cyclic prefix (CP) is added before each OFDM symbol for collecting the multipath associated with the previous symbol. For data symbols, 192 sub-carriers are data bearing, 8 are used as pilots and the remaining are used as guard bands at the lower and upper frequency extremes.

WiMax specifies a framed (burst) mode of operation but leaves the definition of the actual frame structure to the individual PHYs. In point-to-multipoint (PMP) mode, the OFDM PHY supports a frame consisting of a downlink (DL) sub-frame and an uplink (UL) sub-frame.

An uplink sub-frame consists of:

- contention intervals scheduled for initial ranging and bandwidth request purposes, and
- one or multiple UL PHY PDUs, each transmitted from a different subscriber station (SS).

An uplink PHY PDU consists of only one burst, which is made up of a short preamble and an integer number of OFDM symbols.

2.1 Uplink Data Preamble

In the uplink, the data preamble consists of an OFDM symbol containing 2 times 128 samples (in the time-domain) preceded by a cyclic prefix whose length is the same as the cyclic prefix in the traffic mode. This is shown in Figure 1.

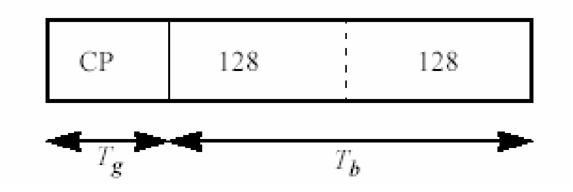

Figure 1 - Uplink Preamble

The preamble symbol is defined in the frequency domain as shown in Table 1. The repeating halves of the preamble symbol (in the time domain) are achieved by setting all odd numbered sub-carriers to null. This effectively makes the data preamble unique as:

- Periods of inactivity are marked by nulls (or AWGN) on all sub-carriers.
- Data symbols will have non-zero values on odd-numbered data sub-carriers.

P(-100:100) = sqrt(2)*sqrt(2)*{	**sub-carrier position**
` 1, 0,-1, 0,-1, 0,-1, 0, 1, 0, 1, 0,`	[-100:-89]
` 1, 0, 1, 0,-1, 0, 1, 0,-1, 0,-1, 0,-1,`	[-88:-76]
` 0, 1, 0,-1, 0, 1, 0, 1, 0, 1, 0, 1,`	[-75:-64]
` 0,-1, 0, 1, 0, 1, 0, 1, 0,-1, 0, 1, 0,`	[-63:-51
`-1, 0, 1, 0, 1, 0,-1, 0,-1, 0, 1, 0,`	[-50:-39]
`-1, 0, 1, 0,-1, 0, 1, 0, 1, 0,-1, 0, 1,`	[-38:-26]
` 0, 1, 0,-1, 0,-1, 0,-1, 0, 1, 0,-1,`	[-25:-14]
` 0,-1, 0,-1, 0,-1, 0,-1, 0, 1, 0, 1, 0,`	[-13:-1]
` 0,`	[0]
` 0, 1, 0,-1, 0,-1, 0, 1, 0,-1, 0, 1, 0,`	[1:13]
` 1, 0, 1, 0, 1, 0,-1, 0, 1, 0, 1, 0,`	[14:25]
` 1, 0, 1, 0,-1, 0, 1, 0,-1, 0,-1, 0,-1,`	[26:38]
` 0,-1, 0, 1, 0, 1, 0,-1, 0, 1, 0,-1,`	[39:50]
` 0,-1, 0,-1, 0,-1, 0,-1, 0,-1, 0,-1, 0,`	[51:63]
`-1, 0, 1, 0, 1, 0, 1, 0,-1, 0,-1, 0,`	[64:75]
`-1, 0, 1, 0, 1, 0,-1, 0,-1, 0,-1, 0, 1,`	[76:88]
` 0,-1, 0,-1, 0, 1, 0,-1, 0,-1, 0,-1}`	[89:100]

Table 1 - Sub-carrier values for UL preamble

The uniqueness of the uplink data preamble and its repetitive structure (in the time-domain) simplify the task of detecting the start of each uplink burst. Symbol timing can be achieved by correlating received samples separated by ½ an OFDM symbol. This is discussed in more detail in the next section.

2.2 Receiver Synchronization

Achieving receiver synchronization for OFDM involves detecting the start of each OFDM symbol and detecting and correcting any frequency offsets in the received signal. Detecting and correcting any frequency offset is especially important for OFDM as even a small offset can result in the orthogonality between sub-carriers being destroyed at the output of the FFT. Due to the presence of the CP, the requirements for symbol timing appear less stringent. The symbol can be deemed to start at any point within the CP not affected by multipath from the previous symbol. An early symbol start manifests as a phase rotation in the sub-carrier values at the output of the FFT. However, an early symbol start can degrade the equalization performance.

Coarse symbol timing can be obtained in the BS receiver by exploiting the repetitive structure of the uplink data preamble. For a (1x sampled) received signal *r(k)*, two correlation values are calculated on a sample-by-sample basis:

Equation 1

$$P(k) = \sum_{n=k-255}^{n=k-128} r(n) \times r^{*}(n+128) \quad E(k) = \sum_{n=k-255}^{n=k} r(n) \times r^{*}(n)$$

In the absence of noise and sampling errors:

Equation 2

$$\begin{aligned} v(k) &= \frac{2 \times |P(k)|}{|E(k)|} \\ &= 1 \qquad \text{(at the end of the preamble symbol)} \\ &\approx 0 \qquad \text{(elsewhere)} \end{aligned}$$

The presence of the CP will actually cause *v(k)* to reach 1 for several samples before the actual end of the preamble symbol. The plateau in *v(k)* makes it difficult to determine the symbol timing precisely, hence the name coarse symbol timing. Selecting the coarse symbol timing so that it errs towards an early symbol start (i.e. within the CP) is acceptable for the reason given above. Fine symbol timing can be performed in the frequency domain to locate the symbol start more accurately. This, however, is beyond the scope of this application note. Noise and sampling errors (such as aperture jitter) will result in *v(k)* falling short of 1 at the end of the preamble.

A frequency offset may comprise of; (a) an integer number of sub-carrier spacings plus (b) a fraction of a sub-carrier spacing. The fractional frequency offset needs to be corrected before the received signal can be processed by the FFT. The fractional frequency offset can be determined from the value of *P(k)* selected by coarse symbol timing.

$$\text{angle}\big(P(k)\big)\,\text{(radians)} \propto \text{fractional frequency offset (Hz)}$$

3 The PC102 picoArray^{TM}

An overview of the salient features of the PC102 picoArray^{TM} is given here in order to understand the OFDM receiver synchronization block described in section 4. Reference is made to the source code given in section 5.2 for the receive buffer for illustrative purposes.

The PC102 consists of an array of processors or array elements interconnected by a high-speed switching fabric called the picoBus.

3.1 Array Elements

The PC102 contains four different types of array elements (AEs) which are detailed in Table 2. Minor differences exist between the three programmable AE types (STAN2, MEM2 and CTRL2). These differences include the size of instruction/data memory, additional processing unit and instructions supported (e.g. multiply-accumulate, multiply). A long instruction word (LIW) of upto 64bits allows upto 3 execution units to be targeted in a single cycle (160MHz). Each AE has a number of ports for communicating with other AEs within the array.

In addition to the STAN2, MEM2 and CTRL2 AE types specified in Table 2, software for the PC102 can also be targeted at the ANY2 AE type implying that:

- the function does not use any AE-specific instructions, and
- the code and data memory requirements can be met by all AE types.

Type	**Description**	**Number**	**Memory (Bytes)**
STAN2	Standard A standard AE type includes multiply-accumulate peripheral as well as special instructions optimized for CDMA spread & de-spread. Memory is divided between 512 bytes code and 256 bytes data.	240	768
MEM2	Memory An AE having multiply unit and additional memory. Memory division between code and data is configurable.	64	8,704
FAU	Function Accelerator Unit A co-processor optimised for specific signal processing tasks (FEC, preamble detect, FHT, etc). Includes dedicated hardware for trellis operations.	14	n/a
CTRL2	Control An AE type with a multiply unit and larger amounts of data and instruction memory optimized for the implementation of basestation control functionality. Memory division between code and data is configurable.	4	65,536
	Totals per PC102 device:	**322**	**1,003,520**

Table 2: PC102 processor variants and memory distribution

Software, written in C or ASM, is targeted at an AE type depending on the processing units used and memory required. Section 5.2 shows the ASM source between the *code* (line 55) and *endcode* (line

68) tags. The C or ASM code for each AE is contained within a picoVHDL wrapper which defines the ports and the type of AE used amongst other things. In section 5.2, line 36 (*begin MEM*) tells us that this code is targeted at a MEM type AE.

NOTE: The MAC, STAN, MEM, CTRL and ANY AE types are also supported on the PC102 for backwards compatibility with the PC101. Where an AE code body does not use any of the additional features which are specific to the PC102, using these AE types allows the s/w to run on both the PC101 and PC102.

3.2 picoBus Switching Fabric

The picoBus is the name given to the switching fabric running vertically and horizontally between the processing elements in the array. Signals (between AEs) are assigned 32-bit slots on the picoBus at compile time thereby removing the need for arbitration and making performance completely deterministic.

Each AE communicates over the picoBus via its ports. These are defined using picoVHDL. Each AE has a number of ports which can be configured to be read (incoming) or write (outgoing). Lines 28-31 in section 5.2 provide an example. Data sent between AEs is:

- written to a write port FIFO (by the sending AE),
- sent over the picoBus on the next available slot and
- read from the read port FIFO (by the receiving AE).

By default, communication between AEs is data blocking. On attempting to read data from the picoBus, an AE will block until data becomes available in the read port FIFO. Similarly, when attempting to write data to the picoBus, the sending AE will block if its write port FIFO is full. A full write port FIFO infers that the receiving AE's read port is not taking data (i.e. is full itself).

Bandwidth on the picoBus between communicating AEs is assigned via @-rates. A signal is assigned an @-rate which is a power of 2, e.g. @8, @16. The @-rate is defined in the port declarations in both the sending and receiving AEs (see lines 29-31 in section 5.2). This @-rate is relative to the system clock (160MHz) and indicates how often data may be sent. For example, @8 means that a 32-bit quantity can be sent every 8 cycles (of the 160MHz bus). The receiving AE(s) must therefore issue a read (against the associated port) at least once every 8 cycles in order to prevent the sending AE from blocking.

4 Uplink Preamble Detection on the picoArray^{TM}

The picoArray^{TM} architecture enables parallelisms within an algorithm to be exploited, resulting in a level of performance associated with FPGA/ASICs whilst maintaining all the benefits associated with a software development environment.

This section describes a PC102 picoArray^{TM} implementation of a BS receiver synchronization block for the WirelessMAN-OFDM uplink. Coarse symbol timing is extracted from the received signal samples. The implementation is validated against a simple MATLAB model. The resource and performance characteristics for the implemented block are summarized below. A full source listing for the block is given in section 5.

Input	≤10 Msps, 16+j16
AE Resources[1]	3 MEM, 2 STAN2

[1] The MEM AE type is used as no PC102 specific features are required.

Table 3 - PC102 OFDM Rx Synchronization

4.1 Preamble Detection

As discussed in section 2.2, the coarse symbol timing can be determined from the data preamble used in the OFDM PHY uplink. Rather than calculating *v(k)* as in Equation 2, *v(k)* is calculated from the magnitude squared values of the two correlation values as shown in Equation 3. This avoids having to calculate the magnitude of the complex value *P(k)*.

Equation 3
$$v(k) = \frac{|P(k)|^2}{|E(k)|^2}$$

The coarse symbol timing is determined from the per-sample value of *v(k)* as follows:

a) As *v(k)* goes above a threshold value, a counter is started and incremented per sample.
b) The counter continues until *v(k)* drops below the threshold value for 5 consecutive samples.
c) Once the counter is stopped, it is compared against a threshold value (64). If greater or equal to this threshold value, a preamble symbol is deemed to have been received.
d) The symbol start is calculated as being ¼ of the way between the counter start and stop samples. This should mean the symbol timing errs towards an early start to the symbol (i.e. within the CP).

The values for *P(k)* and *E(k)*, as used in Equation 3, are calculated iteratively as follows:

$$P(k) = P(k-1) + \left(r^*(k) \times r(k-128)\right) - \left(r^*(k-128) \times r(k-256)\right)$$

$$E(k) = E(k-1) + \left(r^*(k) \times r(k)\right) - \left(r^*(k-256) \times r(k-256)\right)$$

From the above, it becomes apparent that a certain amount of buffering is required:

- The received samples, *r(k)*, need to be buffered in order to supply the delayed (by ½ OFDM symbol) samples for calculating *P(k)*. The received samples also need to be buffered due to the processing delay in determining coarse symbol timing.
- The per-sample values used in the iterative calculations for *P(k)* and *E(k)* are buffered to avoid having to recalculate them.
- The values of *P(k)* need to be buffered for the period that *v(k)* is above the detection threshold. Once coarse symbol timing is determined, the fractional frequency offset can be determined from the appropriate *P(k)*.

4.2 picoArray™ Implementation

The picoArray™ PC102 implementation for the receiver synchronization block is organized as a top-level structural entity and a number of AE code entities. This is illustrated in Figure 2.

The top-level structural entity (*Sync*, section 5.1) uses picoVHDL to 'wire' the various AE code entities together. The structural entity effectively encapsulates the AE code entities contained within it and defines an external interface through which the block is used. The structural entity (*Sync*) may then be instantiated one or more times at a higher-level within the overall design.

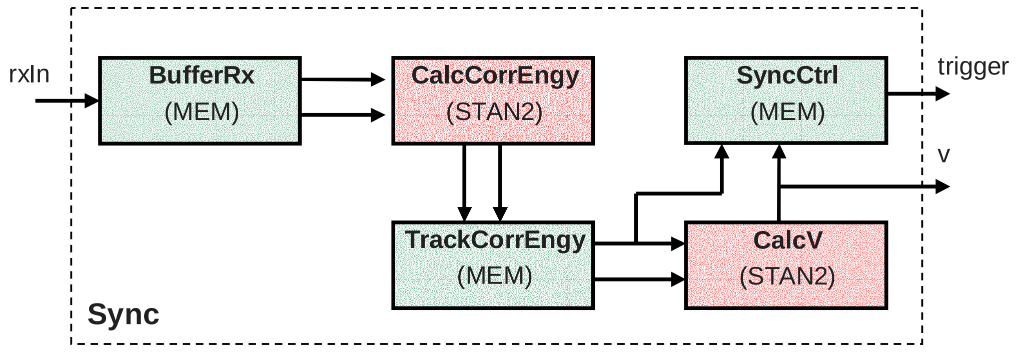

Figure 2 – 'Sync' structural entity

Each AE code entity in Figure 2 runs concurrently and synchronously with respect to the system clock (160MHz). In order to perform at upto 10Msps, the per-sample signals between the AEs need to be @16 or faster (e.g. @8). Each AE must also perform its per-sample processing in 16 cycles or less. The AE code entities are described in more detail in Table 4.

From the AE entity descriptions in Table 4, the *CalcV* entity is the limiting factor with taking 15 cycles per sample. To increase the performance significantly above 10Msps, the processing in *CalcV* could be split between two AEs, each taking <<15 cycles to process each sample. This is an example of the deterministic scalable performance offered by the picoArray™ – performance is increased by pipelining and exploiting parallelisms within an algorithm.

The *CalcCorrEngy*, *TrackCorrEngy* and *CalcV* entities could be left running for each and every received sample without any impact on the performance of other algorithms running on the picoArray^{TM}. Alternatively, control could be added to the *RxBuffer* entity which would cause samples to be sent only when synchronization is being sought. This would cause the *CalcCorrEngy*, *TrackCorrEngy* and *CalcV* entities to block, resulting in a power saving. With only 8 cycles currently in the core loop of *RxBuffer*, this control logic could easily be added without impacting the target 10Msps performance.

AE Code Body	**Description**
BufferRx	The incoming signal samples are buffered in memory. A MEM AE is used so that several OFDM symbols worth of samples may be buffered to account for the overall processing delay of the synchronization block. The current and delayed (buffered) samples are reduced to 8+j8 and then sent to the *CalcCorrEngy* AE. The code body (lines 58-68, section 5.2) uses 8 cycles in its per-sample loop. As such, *BufferRx* could either cope with a faster sample rate or additional functionality could be added (see section 4.3).
CalcCorrEngy	The current and delayed signal samples (8+j8) are taken as inputs and the values for $r^*(k)r(k)$ and $r^*(k)r(k-128)$ generated as outputs. Use of multiply-accumulate instructions necessitate the use of a STAN2. The code body (lines 58-70, section 5.3) uses 10 cycles in its core loop.
TrackCorrEngy	The per-sample outputs from *CalcCorrEngy* are taken and buffered. These values are used to iteratively calculate the running totals for *P(k)* and *E(k)*. These totals are output on a per-sample basis. The buffers for *P(k)* and *E(k)* necessitate the use of a MEM. The code body (lines 63-89, section 5.6) has 12 cycles in its core loop.
CalcV	The per-sample (32-bit) values for *P(k)* and *E(k)* are taken as inputs. *E(k)* and *P(k)* are reduced to 16-bit values. A lookup table is used to approximate the division of *P(k)* by *E(k)*. A 32-bit value for *v* is generated as an output on a per-sample basis. Use of multiply-accumulate instructions necessitate the use of a STAN2. The code body (lines 93-131, section 5.4) has 15 cycles in its core loop (irrespective of the branches taken).
SyncCtrl	The per-sample values for *v(k)* and *P(k)* are taken as inputs. The detection algorithm detailed above is used to extract the coarse symbol timing. The values for *P(k)* are buffered so that the fractional frequency offset can be calculated (not actually performed). The amount of buffering necessitates the use of a MEM. The code body (lines 65-92, section 5.5) contains 10 cycles in the per-sample processing loop until the point at which a preamble symbol is deemed to have been received.

Table 4 - Entities within 'Sync'

4.3 A Complete Synchronization Block

The source code presented in section 5 and described above implements the extraction of coarse symbol timing for the OFDM PHY uplink. Full receiver synchronization also involves detecting frequency offsets and making fine timing adjustments as determined from frequency-domain processing. Frequency offset correction is assumed to be performed by a separate NCO and complex multiplier.

As mentioned in the previous section, additional control logic could be added to the *RxBuffer* entity without affecting the 10Msps performance target. The additional control logic could be used to:

- Send samples to the *CalcCorrEngy*, *TrackCorrEngy* and *CalcV* entities only when burst synchronization is being sought.
- Remove the CP and output the OFDM symbol samples to the next block in the receiver chain (the frequency offset correction block).
- Apply fine timing adjustments as directed.

With being implemented on a MEM, the code and data memory available should easily be enough for the above additional tasks.

Similarly, the *SyncCtrl* entity is also significantly under-utilized. Calculating the fractional frequency offset could be performed in this AE – indeed, that's why the *P(k)* values are buffered in the current implementation.

4.4 Test Results

The PC102 implementation described in this application note was run in the cycle-accurate picoTools simulator to verify its functionality. The test vectors and results are read from and written to file by the simulator. The results where compared against a simple MATLAB model of the algorithm presented in section 4.1. The normalized results from both the picoArray™ implementation and the MATLAB model are shown in Figure 3.

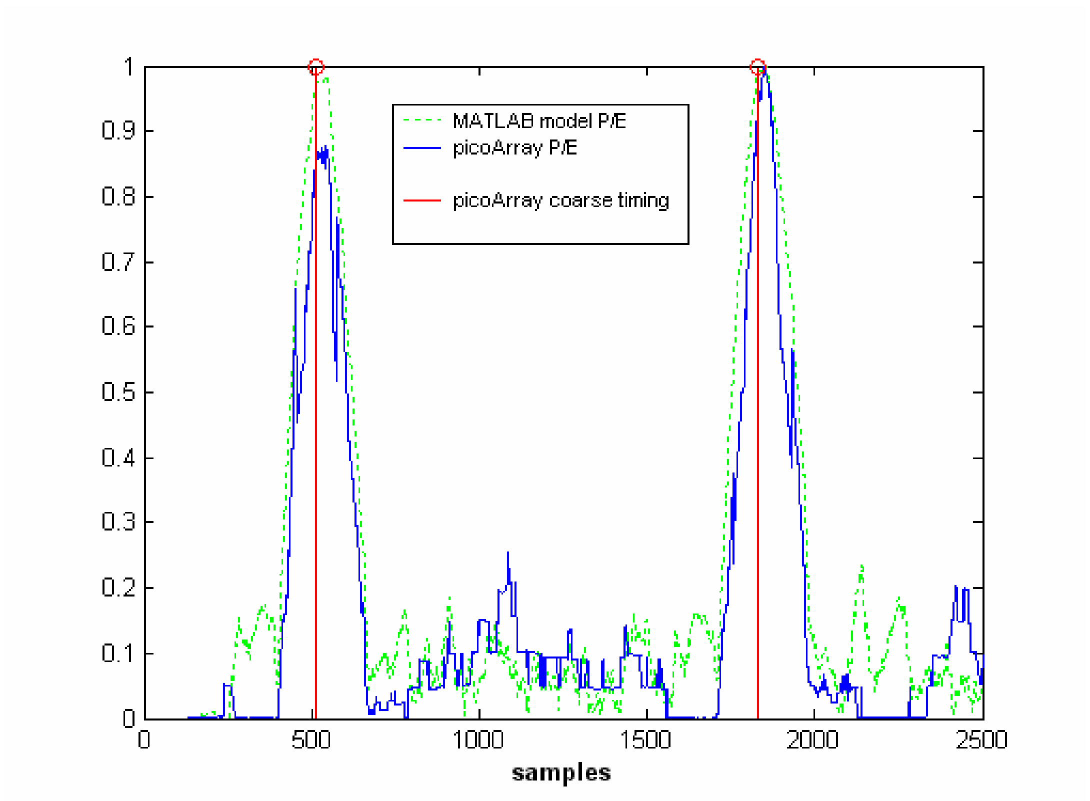

Figure 3 - picoArray preamble detection vs MATLAB model

The test inputs used for both the picoArray^{TM} implementation and MATLAB model consisted of:

- 1x sampled data
- 20-tap channel filter for simulating multipath
- 1/8 CP (32 samples)
- Preamble symbols interspersed by data symbols
- Rx ADC sampling jitter

The red line in Figure 3 indicates the coarse symbol timing determined by the picoArray^{TM} implementation.

5 Appendix: Source Code

5.1 Sync

```
---------------------------------------------------------------------------------
-- Sync
---------------------------------------------------------------------------------
--
-- Copyright (c) 2003 picoChip Designs Ltd.
-- Proprietary and Confidential Information.
-- Not to be copied or distributed.
--
---------------------------------------------------------------------------------
-- Description:
--*****************************************************************************
--*  @short Structural entity for sync block
--*
--*  The structural entity uses picoVHDL to wire the functional entities
--*  together.
--*
--*  @port rxIn     received sample
--*  @port v        pre sample value for v (correlation^2/energy^2)
--*  @port trigger  signal sent when preamble detected
--*****************************************************************************
--/
use work.all;

entity Sync is
  generic(
    bitwidth : integer);
  port(
    rxIn    : in  complex16@16;
    v       : out integer32@16;
    trigger : out integer16pair@16);
end entity Sync;

architecture STRUCTURAL of Sync is

  signal rxDel : complex16@16;
  signal rxCur : complex16@16;
  signal corr  : complex16@16;
  signal engy  : complex16@16;
  signal corrTotal  : complex16@16;
  signal engyTotal  : complex16@16;

begin

  bufRx : entity BufferRx
    generic map (
      bitwidth   => bitwidth )
    port map (
      rxIn       => rxIn,
      rxDel      => rxDel,
      rxCur      => rxCur );

  trackPreamble : entity TrackCorrEngy
    port map (
      corr       => corr,
      engy       => engy,
      corrTotal  => corrTotal,
      engyTotal  => engyTotal );

  calcPreamble : entity CalcCorrEngy
    port map (
      rxDel => rxDel,
      rxCur => rxCur,
      corr  => corr,
      engy  => engy );

  calcPoverE : entity CalcV
    port map (
      corrTotal  => corrTotal,
      engyTotal  => engyTotal,
      v          => v );

  preambleSync : entity SyncCtrl
    port map (
      v         => v,
      corrTotal => corrTotal,
      trigger   => trigger );

end Sync;
```

5.2 BufferRx

```
---------------------------------------------------------------------------------
-- BufferRx
---------------------------------------------------------------------------------
--
-- Copyright (c) 2003 picoChip Designs Ltd.
-- Proprietary and Confidential Information.
-- Not to be copied or distributed.
--
---------------------------------------------------------------------------------
-- Description:
--*****************************************************************************
--*  @short Buffer (filtered) receive samples.
--*
--*  After filtering, the receive samples are buffered in order to perform
--*  preamble detection.
--*
--*  @generic bitwidth  bitwidth of received samples (eg. 10 if 10+j10)
--*
--*  @port rxIn         Received samples to be buffered
--*  @port rxDel        Delayed sample, 1/2 an OFDM symbol behind
--*  @port rxCur        Current received sample
--*****************************************************************************
--/

entity BufferRx is
  generic(
    bitwidth   : integer);
  port(
    rxIn       : in  complex16@16;
    rxDel      : out complex16@16;
    rxCur      : out complex16@16);
end entity BufferRx;

architecture ASM of BufferRx is

begin MEM  -- Uses a MEMORY AE

-- Buffer 3 symbols worth (at 1x) plus CPs
initialize memory  0 : array(0 to 1023) of integer32 := (others => 0);

 -- Initialisation of Registers to zero (apart from delayed sample index)
initialize regs := (0,0,128,0,0,0,0,0,0,0,0,0,0,0);

--Register Definitions.
register rxPtr        is r0;    -- Byte address for base of received sample buffer
register rxDelIdx     is r1;    -- Index into buffer for delayed sample (1/2 sym)
register rxCurIdx     is r2;    -- Index into buffer for current sample
register rxCurRe      is r4;    -- Current sample (real part)
register rxCurIm      is r5;    -- Current sample (imag part)
register rxDelRe      is r6;    -- Delayed sample (real part)
register rxDelIm      is r7;    -- Delayed sample (imag part)
register rxNormRe     is r8;    -- Normalized (to 8+j8) sample (real part)
register rxNormIm     is r9;    -- Normalized (to 8+j8) sample (imag part)

code
    get rxIn,[rxCurIm:rxCurRe]

top:
    lsl.0 rxDelIdx,2,rxPtr
    add.0 rxDelIdx,1,rxDelIdx         \ ldl (rxPtr)0,[rxDelIm:rxDelRe]
    asr.0 rxCurIm,bitwidth-8,rxNormIm \ asr.1 rxCurRe,bitwidth-8,rxNormRe
    and.0 rxDelIdx,1023,rxDelIdx      \ put [rxNormIm:rxNormRe],rxCur
    asr.0 rxDelIm,bitwidth-8,rxNormIm \ asr.1 rxDelRe,bitwidth-8,rxNormRe
    lsl.0 rxCurIdx,2,rxPtr            \ put [rxNormIm:rxNormRe],rxDel
    add.0 rxCurIdx,1,rxCurIdx         \ stl [rxCurIm:rxCurRe],(rxPtr)0 \ bra top
=>  and.0 rxCurIdx,1023,rxCurIdx      \ get rxIn,[rxCurIm:rxCurRe]
```

```

endcode;
end BufferRx;
```

5.3 CalcCorrEngy

```
----------------------------------------------------------------------------------
-- CalcCorrEngy
----------------------------------------------------------------------------------
--
-- Copyright (c) 2003 picoChip Designs Ltd.
-- Proprietary and Confidential Information.
-- Not to be copied or distributed.
--
----------------------------------------------------------------------------------
-- Description:
--*****************************************************************************
--*  @short Calculate correlation and energy values for sample.
--*
--*  The correlation is defined as (a-jb)(c+jd), where (a+jb) is the current
--*  sample and (c+jb) is the delayed sample.
--*  NOTE: The conjugate of the current sample is used in the correlation.
--*
--*  The energy is defined as (a+jb)(a-jb).
--*
--*  The delayed and current sample bitwidth <=8
--*
--*  @port rxDel   Delayed sample
--*  @port rxCur   Current sample
--*  @port corr    Correlation value (complex)
--*  @port engy    Energy value (real)
--*****************************************************************************
--/

entity CalcCorrEngy is
  port(
    rxDel : in  complex16@16;
    rxCur : in  complex16@16;
    corr  : out complex16@16;
    engy  : out complex16@16);
end entity CalcCorrEngy;

architecture ASM of CalcCorrEngy is

begin STAN2  -- Uses a STAN2 AE

 -- Initialisation of Registers to zero
initialize regs := (0 to 14 => 0);

--Register Definitions.
register rxDelRe      is r0;   -- Delayed sample for correlation (real)
register rxDelIm      is r1;   -- Delayed sample for correlation (imag)
register rxCurRe      is r2;   -- Current sample (real)
register rxCurIm      is r3;   -- Current sample (imag)
register corrRe       is r4;   -- correlation (real)
register corrIm       is r5;   -- correlation (imag)
register engyRe       is r6;   -- energy (real)
register engyIm       is r7;   -- energy (imag=zero)

code
    get rxCur,[rxCurIm:rxCurRe]
    get rxDel,[rxDelIm:rxDelRe] \ mul rxCurRe,rxCurRe,acc0

top:
    mac rxCurIm,rxCurIm,acc0
    readacc acc0,frac,engyRe
    put [engyIm:engyRe],engy    \ mul rxCurRe,rxDelRe,acc0
    mac rxCurIm,rxDelIm,acc0  -- NB: Multiplying delayed by conjugate of current
    readacc acc0,frac,corrRe
    mul rxCurRe,rxDelIm,acc1
    msub rxCurIm,rxDelRe,acc1 -- NB: Multiplying delayed by conjugate of current
    readacc acc1,frac,corrIm
    put [corrIm:corrRe],corr    \ get rxCur,[rxCurIm:rxCurRe] \ bra top
=-> get rxDel,[rxDelIm:rxDelRe] \ mul rxCurRe,rxCurRe,acc0

endcode;
end CalcCorrEngy;
```

5.4 CalcV

```
--------------------------------------------------------------------------------
-- CalcV
--------------------------------------------------------------------------------
--
-- Copyright (c) 2003 picoChip Designs Ltd.
-- Proprietary and Confidential Information.
-- Not to be copied or distributed.
--
--------------------------------------------------------------------------------
-- Description:
--*****************************************************************************
--*  @short Calculate V = correlation/energy.
--*
--*  As the preamble consists of 2 repeating halfs (in the time domain),
--*  2*correlation/energy should approach 1 as the current sample reaches the
--*  end of the preamble symbol.
--*
--*  This block approximates 2*correlation/energy by correlation^2/energy^2.
--*  The theoretical max value for correlation^2 is 1/4*energy^2.
--*
--*  The energy total (squared) is normalized to the range 2^15 to 2^16-1.
--*  The correlation total (squared) is normalized by shift by the same number
--*  of bits minus 2 (given the max value is 1/4 of energy^2).
--*
--*  A lookup table is used to perform the division.  The energy^2 value is
--*  reduced to 8 bits (bottom 2 bits = 0) to form an index into the table.
--*
--*  At the point of detection correlation^2 should approach energy^2 (due to
--*  effective multipication by 4 in the shifting above).
--*  Therefore, taking energy=2^16-1 as an example, the last entry in the
--*  division LUT is used, giving a result of 2^16-1 * 8224 = 5.39e8.
--*  v should approach 5.39e8 at the point of detection for all values of
--*  energy.
--*
--*  A 32-bit value for v is returned.
--*
--*  @port corrTotal  Correlation total (summed over half OFDM symbol)
--*  @port engyTotal  Energy total(summed over 1 OFDM symbol)
--*  @port v          correlation^2/energy^2
--*****************************************************************************
--/
entity CalcV is
  port(
    corrTotal  : in  complex16@16;
    engyTotal  : in  complex16@16;
    v          : out integer32@16);
end entity CalcV;

architecture ASM of CalcV is

begin STAN2

-- Lookup table for dividing in the range 1/128 to 1/255
initialize memory 0 : array(0 to 127) of integer16 := (
  16384,16257,16132,16009,15888,15768,15650,15534,
  15420,15308,15197,15087,14980,14873,14769,14665,
  14564,14463,14364,14266,14170,14075,13981,13888,
  13797,13707,13618,13530,13443,13358,13273,13190,
  13107,13026,12945,12866,12788,12710,12633,12558,
  12483,12409,12336,12264,12193,12122,12053,11984,
  11916,11848,11782,11716,11651,11586,11523,11460,
  11398,11336,11275,11215,11155,11096,11038,10980,
  10923,10866,10810,10755,10700,10645,10592,10538,
  10486,10434,10382,10331,10280,10230,10180,10131,
  10082,10034,9986,9939,9892,9846,9800,9754,
  9709,9664,9620,9576,9533,9489,9447,9404,
  9362,9321,9279,9239,9198,9158,9118,9079,
  9039,9001,8962,8924,8886,8849,8812,8775,
  8738,8702,8666,8630,8595,8560,8525,8490,
  8456,8422,8389,8355,8322,8289,8257,8224
);

 -- Initialisation of Registers to zero
initialize regs := (0 to 14 => 0);

--Register Definitions.
register corrRe       is r0;   -- correlation (real)
register corrIm       is r1;   -- correlation (imag)
register engyRe       is r2;   -- energy (real)
register engyIm       is r3;   -- energy (imag=zero)
register corr2Lo      is r4;   -- correlation squared (lower 16-bits)
register corr2Hi      is r5;   -- correlation squared (upper 16-bits)
register engy2Lo      is r6;   -- energy squared (lower 16-bits)
register engy2Hi      is r7;   -- energy squared (upper 16-bits)
register signBits     is r8;   -- num of (redundant) sign bits in energy
register dataBits     is r9;   -- number of data bits
register msbIsData    is r10;  -- MSB in lower 16-bits is data (and not sign)
register divIdx       is r11;  -- Index into division LUT
register vLo          is r12;  -- v (lower 16-bits)
register vHi          is r13;  -- v (upper 16-bits)
register xfactor      is r14;  -- multiplication factor from division LUT

code
    get engyTotal,[engyIm:engyRe]
    mul engyRe,engyRe,acc0

top:
    get corrTotal,[corrIm:corrRe] \ readacc32 acc0,[engy2Hi:engy2Lo]
    sbc engy2Hi,signBits \ mul corrRe,corrRe,acc0
    sub.0 signBits,15,r15 \ mac corrIm,corrIm,acc0
    lsr.0 engy2Lo,15,msbIsData \ beq lowerBitsOnly
=> readacc32 acc0,[corr2Hi:corr2Lo]

upperBits:
    -- Energy contains significant bits in upper 16-bit word
    -- Shift so that MSB is bit14 (0 to 15) of lower 16-bit word
    lsl.0 engy2Hi,signBits,engy2Hi \ sub.1 16,signBits,dataBits
    lsr.0 engy2Lo,dataBits,engy2Lo \ lsl.1 corr2Hi,2,corr2Hi
    lsl.0 corr2Hi,signBits,corr2Hi \ or.1 engy2Hi,engy2Lo,engy2Lo
    and.0 [lsr engy2Lo,7],16#fc#,divIdx \ lsr.1 corr2Lo,2,corr2Lo
    lsr.0 corr2Lo,dataBits,corr2Lo \ ldw (divIdx)0,xfactor \ bra compare
=>  or.0 corr2Hi,corr2Lo,corr2Lo

lowerBitsOnly:
    -- Energy only contains significant bits in lower 16-bit work
    -- Shift so that MSB is bit14 (0 to 15) of lower 16-bit word
    lsr.0 engy2Lo,msbIsData,engy2Lo
    sbc engy2Lo,signBits
    lsl.0 engy2Lo,signBits,engy2Lo
    and.0 [lsr engy2Lo,7],16#fc#,divIdx \ lsl.1 corr2Lo,2,corr2Lo
    lsr.0 corr2Lo,msbIsData,corr2Lo \ ldw (divIdx)0,xfactor \ bra compare
=> lsl.0 corr2Lo,signBits,corr2Lo

compare:
    -- MSB of energy should be after sign bit
    mul corr2Lo,xfactor,acc0
    readacc32 acc0,[vHi:vLo]
    put [vHi:vLo],v \ get engyTotal,[engyIm:engyRe] \ bra top
=> mul engyRe,engyRe,acc0

endcode;
end CalcV;
```

5.5 SyncCtrl

```
--------------------------------------------------------------------------------
-- SyncCtrl
--------------------------------------------------------------------------------
--
-- Copyright (c) 2003 picoChip Designs Ltd.
-- Proprietary and Confidential Information.
-- Not to be copied or distributed.
--
--------------------------------------------------------------------------------
-- Description:
--*****************************************************************************
--*  @short Synchronization control
--*
--*  Coarse symbol timing is determined for the value for v (1 per rx sample).
--*  Coarse timing is calculated by waiting for v to go above the threshold
--*  value for a certain number of samples.  As v falls back down below the
--*  threshold then the coarse sample timing is taken as 1/4 of period
--*  v was above the threshold.
--*
--*  Once coarse timing has been calculated, the initial fractional frequency
--*  offset can be detemined (not done here).
--*
--*  @port v           Approximation of 2*correlation/energy
--*  @port corrTotal   Running correlation total (per sample)
--*  @port trigger     Trigger when coarse timing determined
--*****************************************************************************
--/
entity SyncCtrl is
  port(
    v         : in  integer32@16;
    corrTotal : in  complex16@16;
    trigger   : out integer16pair@16);
end entity SyncCtrl;

architecture ASM of SyncCtrl is

-- Threshold level taken from upper 16-bits of v.
constant VLEVEL : integer := 16#0800#;
-- Min number of samples needed above detection level for sync
constant VABOVE : integer := 64;
-- Number of samples v is allowed to dip below level without resetting
constant VBELOW : integer := 5;

begin MEM

-- Buffer correlation totals
initialize memory  0 : array(0 to 256) of integer32 := (others => 0);

 -- Initialisation of Registers to zero
initialize regs := (0 to 14 => 0);

--Register Definitions.
register vLo            is r0;  -- v (lower 16-bits)
register vHi            is r1;  -- v (upper 16-bits)
register corrTotalRe    is r2;  -- running correlation total (real)
register corrTotalIm    is r3;  -- running correlation total (imag)
register corrPtr        is r4;  -- pointer into correlation buffer
register belowCtr       is r5;  -- number of consecutive samples below threshold
register nDetect        is r6;  -- number of samples detected above threshold
register nSamples       is r7;  -- number of samples processed
register triggerLo        is r8;   -- sample number corresponding to the point of
detection
register triggerHi      is r9;  -- not used

code

```

```
top:
    add.0 nSamples,1,nSamples \ get v,[vHi:vLo]
    sub.0 vHi,VLEVEL,r15 \ get corrTotal,[corrTotalIm:corrTotalRe]
    stl [corrTotalIm:corrTotalRe],(corrPtr)0 \ blt below  -- branch if below level
=-> sub.0 [lsr corrPtr,2],0,r15

above:
    copy.0 0,belowCtr \ bra top
=-> add.1 corrPtr,4,corrPtr

below:
    add.0 belowCtr,1,belowCtr \ beq top  -- branch if no recent samples above level
=-> sub.0 belowCtr,VBELOW,r15
    blt top  -- branch if still within limit of samples below level
=-> sub.0 [lsr corrPtr,2],VABOVE,r15
    sub.0 [lsr corrPtr,2],VBELOW,nDetect \ blt top -- branch if not enough samples
=-> copy.0 0,corrPtr

detect:
    -- We've detected enough samples above the level, work out coarse timing
    sub.0 nSamples,nDetect,triggerLo \ lsr.1 nDetect,2,nDetect
    add.0 triggerLo,nDetect,triggerLo \ bra top
=-> put [triggerHi:triggerLo],trigger


endcode;
end SyncCtrl;
```

5.6 TrackCorrEngy

```
----------------------------------------------------------------------------------
-- TrackCorrEngy
----------------------------------------------------------------------------------
--
-- Copyright (c) 2003 picoChip Designs Ltd.
-- Proprietary and Confidential Information.
-- Not to be copied or distributed.
--
----------------------------------------------------------------------------------
-- Description:
--****************************************************************************
--*  @short Keep running totals for correlation and energy over 1 OFDM symbol
--*
--*  A running correlation total is kept for r'(n)*r(n-128) summed over half
--*  an OFDM symbol (128 samples at 1x).
--*  A running energy total is kept for r'(n)*r(n) summed over a full OFDM
--*  symbol.
--*
--*  The per sample correlation and energy values are buffered.  The totals
--*  are calculated iteratively.
--*
--*  @port corr       per sample correlation value
--*  @port engy       per sample energy value
--*  @port corrTotal  current total for correlation (over 1/2 OFDM symbol)
--*  @port engyTotal  current total for energy (over 1 OFDM symbol)
--****************************************************************************
--/
entity TrackCorrEngy is
  port(
    corr       : in  complex16@16;
    engy       : in  complex16@16;
    corrTotal  : out complex16@16;
    engyTotal  : out complex16@16);
end entity TrackCorrEngy;

architecture ASM of TrackCorrEngy is

constant CORR_BASE  : integer := 0;
constant ENGY_BASE  : integer := 512;

begin MEM

initialize memory  CORR_BASE  : array(0 to 127) of integer32 := (others => 0);
initialize memory  ENGY_BASE  : array(0 to 255) of integer32 := (others => 0);

-- Initialisation of Registers to zero
initialize regs := (CORR_BASE,0,ENGY_BASE,0,0,0,0,0,0,0,0,0,0,0,0,0);

--Register Definitions.
register corrPtr      is r0;   -- pointer to entry in buffer of correlation values
register corrIdx      is r1;   -- index into correlation buffer
register engyPtr      is r2;   -- pointer to entry in buffer of energy values
register engyIdx      is r3;   -- indx into energy buffer
register corrRe       is r4;   -- per sample correlation value (real)
register corrIm       is r5;   -- per sample correlation value (imag)
register corrTotalRe  is r6;   -- correlation running total (real)
register corrTotalIm  is r7;   -- correlation running total (imag)
register engyRe       is r8;   -- per sample energy value (real)
register engyIm       is r9;   -- per sample energy value (imag=0)
register engyTotalRe  is r10;  -- energy running total (real)
register engyTotalIm  is r11;  -- energy running total (imag=0)

code
    add.0 [lsl engyIdx,2],ENGY_BASE,engyPtr

top:
    -- For correlation and energy:
    --   o Calculate pointer into buffer
    --   o Read value from buffer - this is the oldest value
    --   o Subtract oldest value from total
    --   o Get latest per-sample value
    --   o Add latest value to total
    --   o Write latest value to buffer and advance index

    add.0 [lsl corrIdx,2],CORR_BASE,corrPtr \ ldl (engyPtr)0,[engyIm:engyRe]
    add.0 engyIdx,1,engyIdx \ ldl (corrPtr)0,[corrIm:corrRe]
    add.0 corrIdx,1,corrIdx \ sub.1 engyTotalRe,engyRe,engyTotalRe
    sub.0 corrTotalRe,corrRe,corrTotalRe \ sub.1 corrTotalIm,corrIm,corrTotalIm
    and.0 engyIdx,255,engyIdx \ get engy,[engyIm,engyRe]
    asr.0 engyRe,6,engyRe \ asr.1 engyIm,6,engyIm
    add.0   engyRe,engyTotalRe,engyTotalRe   \   get   corr,[corrIm,corrRe]   \   put
[engyTotalIm:engyTotalRe],engyTotal
    asr.0 corrRe,6,corrRe \ asr.1 corrIm,6,corrIm
    add.0 corrRe,corrTotalRe,corrTotalRe \ add.1 corrIm,corrTotalIm,corrTotalIm
    and.0 corrIdx,127,corrIdx \ put [corrTotalIm:corrTotalRe],corrTotal
    stl [engyIm:engyRe],(engyPtr)0 \ bra top
=>  add.0 [lsl engyIdx,2],ENGY_BASE,engyPtr \ stl [corrIm:corrRe],(corrPtr)0

endcode;
end TrackCorrEngy;
```