AN-6: The antif.dll Software Library

1. Summary

This application note describes the RockyLogic *antif.dll* software library – a software interface between an application program and an Ant module attached to a USB port on a local or remote computer. Throughout the note, unless otherwise qualified, an *Ant* refers to either an Ant8 or an Ant16 or an Ant18e.

2. Introduction

antif.dll opens up the Ant support software to allow user programmability, including running programs across a network. Typically Ant application software is implemented as three layers:

a. the graphical user interface (GUI) layer, which interfaces to

b. the *antif* interface DLL, the subject of this application note. This deals with the logical control of the Ant and interfaces across a TCP link to

c. a server program which handles low-level control of the Ants. The interface to the server program will be of little interest to most users, but it is described in application note AN-3.

antif.dll has a standard C language interface and can be called from any language which can call functions in C DLLs.

Typically programs using the DLL will go through these stages:

1. Initialize the DLL
2. Connect to the Ant server
3. Find out which Ant units are plugged in
4. Open the selected Ant
5. Set the logic analyzer parameters
6. Tell the Ant to run
7. Query the Ant status until it has triggered and stopped
8. Read the captured data from the Ant
9. Close the Ant
10. Disconnect from the Ant server
11. Free any DLL resources

2.1 Servers

antif.dll needs to interface to a server which drives the Ants over USB links. These options are supported:

- a built-in server. Connecting to the server with a NULL host name will automatically connect to the built-in server.
- a server program running on the same computer. In accordance with the TCP/IP specification, a host name of *127.0.0.1* will automatically select the same computer. On just about every computer, you can get the same effect by connecting to host name *localhost,* though technically this is not guaranteed.
- a server program running on a connection across a network. Set the host name to either the name or the dotted quad address of the computer on which the server is running. For instance *test_lab* or *192.168.1.26.*

If the built-in server is not used, you will need a separate standalone server. The *antserve.exe* program is the same as the built-in server, with the addition of a small control GUI.

3. Library Functions

Basic functions are

- RL_Info returns the basic operating parameters of an Ant module.
- RL_Initialize initializes the DLL .
- RL_Finalize releases resources when the program completes.

Connecting to the server is handled by

- RL_ConnectToServer opens a connection to the server.
- RL_DisconnectFromServer close the connection.
- RL_QueryServerID returns the server's text string ID.

For connecting to an Ant, the functions are

- RL_AntCount reports the number of Ants connected to the server.
- RL_AntInfo gives the name and status of each Ant.
- RL_OpenAnt opens the selected Ant.
- RL_CloseAnt closes the Ant.

Acquisition parameters are set via

- RL_SetHitPattern
- RL_SetXFunction
- RL_SetStateMachineFunction
- RL_SetClockIx
- RL_SetTriggerPos
- RL_SetTimerCounter
- RL_SetThreshold.

They can be read back via

- RL_GetHitPattern
- RL_GetXFunction
- RL_GetStateMachineFunction
- RL_GetClockIx
- RL_GetTriggerPos
- RL_GetTimerCounter
- RL_GetThreshold.

Run the Ant via

- RL_Run starts and stops the logic analyzer.
- RL_GetRunStatus tells whether the analyzer is stopped or still searching for a pattern
- RL_Readback retrieves the captured data from the Ant.

Frequency Counter functions, only applying to the Ant18e, are

- RL_FCRun sets the frequency counter parameters and starts the counter
- RL_FCGetStatus returns the running/finished status for the counters
- RL_FCGetCounts returns the per-channel counter totals.

Miscellaneous functions are

- RL_GetPinStatus will return the status of the pins on the Ant, when an acquisition is not in progress
- RL_GetLastErrorMessage
- RL_Heartbeat reassures the server that the client is still running.
- RL_SetScratchReg stores a 32-bit value to the server.
- RL_GetScratchReg retrieves the scratch register.
- RL_GetClockInfo converts a clock index to numerical and human readable periods.

All the function descriptions should be read in conjunction with the *antif.h* header file, which contains definitions of the constants and data structures used by the library. The return value from every function is RL_OK if successful, otherwise an error code as defined in the header file.

3.1 Information

```
RL_RESULT RL_Info(int AntType, TAntParams* pInfo, int InfoBytes);
```

Parameters

AntType	114 (the 'r' character) for Ant8, 111 ('o') for Ant16, 99 ('c') for Ant18e.
pInfo	pointer to the buffer to receive information about the Ant capabilities

InfoBytes size of the information structure.

Remarks

The *InfoBytes* parameter guards against future software revisions writing outside the pInfo structure. The members of the *TAntParams* structure are defined in the antif.h header file; this structure is extended from time to time as new capabilities are added to the software.

Example

```
TAntParams params;
RL_RESULT retval = RL_Info('r', &params, sizeof(params));
printf("The Ant8 has %d channels\n", params.Channels);
```

3.2 DLL Initialization and Termination

```
RL_RESULT RL_Initialize(RL_HANDLE* h);
```

Parameters

h handle used for subsequent access to the DLL routines.

Remarks

This function allocates internal structures.

Example

```
RL_HANDLE h=NULL;
RL_RESULT retval = RL_Initialize(&h);
char* result = (retval==RL_OK) ? "success" : "fail";
printf("DLL Initialize result: %s\n", result);
```

```
RL_RESULT RL_Finalize(RL_HANDLE h);
```

Parameters

h handle returned from a previous RL_Initialize.

Remarks

This function deallocates internal structures. As a consequence, any open connection via the server to an Ant is closed and an open connection to a server is closed. The handle becomes invalid.

Example

```
RL_RESULT retval = RL_Finalize(h);
char* result = (retval==RL_OK) ? "success" : "fail";
printf("DLL Finalize result: %s\n", result);
```

3.3 Connecting to a Server and Opening an Ant

```
RL_RESULT RL_ConnectToServer(RL_HANDLE h, char *Host, int Port);
```

Parameters

h handle returned from a previous RL_Initialize.

Host name of the TCP/IP target.

Port port on which the server is listening.

Remarks

Typically the connection is to a server running on the same computer, in which case the server's host computer has an IP address of 127.0.0.1 and is conventionally known as *localhost*. In this case *Host* can be set to either *localhost* or *127.0.0.1*.

For connections across a network, set Host to either the name or the dotted quad address of the computer on which the server is running. For instance *test_lab* or *192.168.1.26.*

The *Port* parameter must match the port number with which the server was started. Usually the server is set to listen on the *DEFAULT_PORT.*

Example

```
char Host[] = LOCALHOST;
RL_RESULT retval = RL_ConnectToServer(h, Host, DEFAULT_PORT);
char* result = (retval==RL_OK) ? ”connected” : ”failed”;
printf(“ConnectToServer result: %s\n”, result);
```

```
RL_RESULT RL_DisconnectFromServer(RL_HANDLE h);
```

Parameters

h handle returned from a previous RL_Initialize.

Remarks

Any open connection via the server to an Ant is closed and an open connection to the server is closed. The handle remains valid and can either be used for subsequent operations or released via RL_Finalize().

Example

```
RL_RESULT retval = RL_DisconnectFromServer(h);
char* result = (retval==RL_OK) ? ”success” : ”fail”;
printf(“DisconnectFromServer result: %s\n”, result);
```

```
RL_RESULT RL_QueryServerID(RL_HANDLE h, char*s, int len);
```

Parameters

h handle returned from a previous RL_Initialize.

s character string for the ID

len max length ID string which can be returned, not including any terminating zero.

Remarks

Returns an identifying string from the server. The format of the ID is shown in the following example:

```
antserver,Jan 10 2006,Jan 23 2006,20:08:52
```

The fields are the string *antserver*, a comma, the creation date (i.e. version) of the server,

another comma, the date on which the server started running, another comma, and the time at which the server started running.

Example

```
RL_HANDLE h=NULL;
char str[64];
memset(str, 0, sizeof(str));
RL_RESULT retval = RL_QueryServerID(h, str, sizeof(str)-1);
char* result = (retval==RL_OK) ? str : "fail";
printf("QueryServerID result: %s\n", result);
```

```
RL_RESULT RL_AntCount(RL_HANDLE h, int* pCount);
```

Parameters

h	handle returned from a previous RL_Initialize.
pCount	number of Ants on the server

Remarks

The server reports all the Ants it knows about, whether or not they are plugged in or busy. The server's rule is to put an Ant on its list once it is detected. If the Ant is then unplugged it is marked as missing, but not removed from the list.

Example

```
int Count = 0;
RL_RESULT retval = RL_AntCount(h, &Count);
char str[64];
sprintf(str, "%d", Count);
char* result = (retval==RL_OK) ? str : "fail";
printf("AntCount result: %s\n", result);
```

```
RL_RESULT RL_AntInfo(RL_HANDLE h, int index, char* name, char*
flags);
```

Parameters

h	handle returned from a previous RL_Initialize.
index	must be within the Count returned by AntCount()
name	buffer to receive the Ant's USB ID string – 8 characters.
flags	buffer to receive the Ant's status flags – 3 characters.

Remarks

The status flags are as follows:

Position	*Character and Meaning*
0	r - Ant8 o – Ant16 c - Ant18e
1	f - free b - busy
2	p - present m - missing

Example

```
char name[ANT_NAME_LENGTH+1];
char flags[FLAGS_LENGTH+1];
memset(name,  0, sizeof(name) );
memset(flags, 0, sizeof(flags));
RL_RESULT retval = RL_AntInfo(h, 0, name, flags);

char c = flags[0];
char* mtype = (c=='r') ? "Ant8 " : ((c=='o') ? "Ant16" : "Ant18e" );
c = flags[1];
char* busy  = (c=='b') ? "busy" : ((c=='f') ? "free" : "?" );
c = flags[2];
char* present = (c=='p') ? "present" : ((c=='m') ? "missing" : "?");

char str[64];
sprintf(str, "%s %s %s %s %s", name, flags, mtype, busy, present);
char* result = (retval==RL_OK) ? str : "fail";
printf("AntInfo result: %s\n", result);
```

```
RL_RESULT RL_OpenAnt(RL_HANDLE h, char* name);
```

Parameters

h handle returned from a previous RL_Initialize.

name name returned from a previous RL_AntInfo .

Remarks

Opens a connection to the Ant and sets the sample acquisition parameters to default values.

Example

```
char name[ANT_NAME_LENGTH+1];
...
// fill via RL_AntInfo()
...
RL_RESULT retval = RL_OpenAnt(h, name);
char* result = (retval==RL_OK) ? "success" : "fail";
printf("OpenAnt result: %s\n", result);
```

```
RL_RESULT RL_CloseAnt(RL_HANDLE h);
```

Parameters

h handle returned from a previous RL_Initialize.

Remarks

Close the connection to the Ant, which can then be used by another application.

Example

```
RL_RESULT retval = RL_CloseAnt(h);
char* result = (retval==RL_OK) ? ”success” : ”fail”;
printf(“ CloseAnt result: %s\n”, result);
```

3.4 Setting Acquisition Parameters

Acquisition parameters can only be set after an Ant has been opened. This is so that the software can check the parameters for legality – the Ant8 requires 8-element hit patterns, the Ant16 hit patterns must have 16 elements, the Ant18e hit patterns must have 18 elements.

All acquisition parameters can be read back - each *SetParameter* function is paired with a *GetParameter* function.

```
RL_RESULT RL_SetHitPattern(RL_HANDLE h, int ix, bool AndCombine,
char* Pat);
```

Parameters

h	handle returned from a previous RL_Initialize.
ix	must be within the *PatternRecognizers* count returned by RL_Info().
AndCombine	true = AND the pattern elements, false = OR the elements.
Pat	match pattern – see below.

Remarks

Pat points to a null-terminated character string which has one position for each input to the Ant. The characters determine the input signals which trigger this pattern recognizer. The encoding of the string is follows:

Character	*Input signal is*
-	don't care
0	low
1	high
R	rising
F	falling
E	rising or falling

For instance, the pattern for an Ant8 could be “F---0000” which means that the pattern matches a *falling edge* on channel 0, anything on channels 1 to 3, and *low* on channels 4 to 7.

AndCombine determines how matches on the individual signals are combined. If *AndCombine* is *true*, the pattern as a whole matches if all of the individual channels match. If

AndCombine is *false*, the pattern as a whole matches if just one of the individual channels match. For this scheme to work, *don't care* is a match when *AndCombine* is true; when *AndCombine* is false, *don't care* is not a match.

Example

```
RL_RESULT retval = RL_SetHitPattern(h, 0, true, "F---0000");
```

```
RL_RESULT RL_GetHitPattern(RL_HANDLE h, int ix, bool* pAndCombine,
char* pBuff, int BuffLen);
```

Parameters

h	handle returned from a previous RL_Initialize.
ix	must be within the *PatternRecognizers* count returned by RL_Info().
pAndCombine	receives the stored value of AndCombine.
pBuff	receives the stored value of the match pattern – see RL_SetHitPattern.
BuffLen	length of the buffer pointed to by pBuff.

Remarks

BuffLen must be large enough to accommodate the null-terminated hit pattern – 9 characters for an Ant8, 17 characters for an Ant16, and 19 characters for an Ant18e.

Example

```
char Buff[17];
bool com;
RL_RESULT retval = RL_GetHitPattern(h, 0, &com, Buff, sizeof(Buff));
```

```
RL_RESULT RL_SetXFunction(RL_HANDLE h, int ix, char* Func);
```

Parameters

h	handle returned from a previous RL_Initialize.
ix	must be within the *XFunctions* count returned by RL_Info()
Func	X function string – see below.

Remarks

The Ants have two pattern recognizers – P0 and P1. The matches on P1 and P2 can be combined in an arbitrary way in the *XFunction* combiners. There are two of these combiners – X0 and X1. Func points to a null-terminated character string which defines the combining function. The operators which can be used in the combining functions are as follows:

Character	*Operation*
&	and
|	or
^	xor
!	not

Opening and closing parentheses - (and) - can also be used. Spaces are ignored. Surrounding the entire function with parentheses is conventional but optional.

Typically we set X0 and X1 to simple functions. For instance “(P0)” or “(P1)”. But we can, for instance, set X0 to P0 OR P1: “(P0 | P1)”.

Example

```
RL_RESULT retval = RL_SetXFunction(h, 0, "(P0 ^ P1)" );
```

```
RL_RESULT RL_GetXFunction(RL_HANDLE h, int ix, char* pBuff, int BuffLen);
```

This function is analogous to RL_GetHitPattern()

```
RL_RESULT RL_SetTimerCounter(RL_HANDLE h, int Val, bool TCIsTimer);
```

Parameters

h	handle returned from a previous RL_Initialize.
Val	must be within the *CounterSize* value returned by RL_Info().
TCIsTimer	true: use the Timer/Counter as a timer, false: use as a counter.

Remarks

The Timer/Counter (TC) is automatically loaded in the Idle state and reloaded

- on entering the Hit1 state *or*
- on entering the Hit2 state *or*
- when in the Hit1 state *or*
- when in the Hit2 states.

Exactly when the TC is reloaded is determined by an expression entered by the *RL_SetStateMachineFunction()* function (see below for a description of this function).

TC counts down to zero

- on entering the Hit1 state *or*
- on entering the Hit2 state *or*
- when in the Hit1 state *or*
- when in the Hit2 state

As before, exactly when the TC counts down is determined by an expression entered by the *RL_SetStateMachineFunction()* function. The down count count stops at zero.

Example

```
RL_RESULT retval = RL_SetTimerCounter(h, 25, false );
```

```
RL_RESULT RL_GetTimerCounter(RL_HANDLE h, int* pVal, bool*
pTCIsTimer);
```

Returns the current value loaded into the timer/counter to the integer pointed to by *pVal*. Also sets *pTCIsTimer* to true if the timer/counter is a timer, and to false if it is a counter.

```
RL_RESULT RL_SetStateMachineFunction(RL_HANDLE h, int ix, char* Fnc);
```

Parameters

h	handle returned from a previous RL_Initialize.
ix	must be within the *StateMachineEquations* count returned by RL_Info()
Fnc	function string – see below.

Remarks

The state machines which control the operation of the Ant8 and the Ant16 are described in the help files. Each transition in the state machine is controlled by a function, the basic rule being “make the transition if the function evaluates to true.”

There are eight functions. Each is an arbitrary function or up to four *inputs*, known as I0, I1, I2, and I3.

Unused inputs are always *true*. The external trigger input is not implemented in the Ant8 and the equation input for the external trigger defaults to *true* .

Index	*Function*	*Inputs*
0	Timer/Counter enable	I0: in Hit1 state I1: in Hit2 state I2: entered Hit1 state I3: entered Hit2 state
1	Timer/Counter reload	I0: in Hit1 state I1: in Hit2 state I2: entered Hit1 state I3: entered Hit2 state
2	transition from Hit2 state to Hit1 state	I0: X0 I1: X1 I2: TC is zero I3: unused
3	transition from Hit2 state to Triggered state	I0: X0 I1: X1 I2: TC is zero I3: Trigger Input
4	transition from Hit1 state to Hit2 state	I0: X0 I1: X1 I2: TC is zero I3: unused
5	transition from Hit1 state to Triggered state	I0: X0 I1: X1 I2: TC is zero I3: Trigger Input
6	transition from Search state to Hit1 state	I0: X0 I1: X1 I2: unused I3: unused
7	transition from Search state to Triggered state	I0: X0 I1: X1 I2: unused I3: Trigger Input

Example

```
RL_RESULT retval = RL_SetStateMachineFunction(h, 7, "I0 | I1");
// trigger as soon as there is a match on either X0 or X1
```

```
RL_RESULT RL_GetStateMachineFunction(RL_HANDLE h, int ix, char*
pBuff, int BuffLen);
```

Parameters

h handle returned from a previous RL_Initialize.

ix must be within the *StateMachineEquations* count returned by RL_Info()

pBuff receives the stored value of the function.

BuffLen length of the buffer pointed to by pBuff.

Remarks

BuffLen must be large enough to accommodate the null-terminated function.

Example

```
char Buff[32];
RL_RESULT retval = RL_GetStateMachineFunction(h, 0, Buff, sizeof(Buff));
```

```
RL_RESULT RL_SetClockIx(RL_HANDLE h, int ClkIx);
```

Parameters

h handle returned from a previous RL_Initialize.

ClkIx must be one of the following values:

```
#define K_Sync                    0       /*   Synchronous */
#define K_500MHz                  1       /* 500MHz -  2ns */
#define K_250MHz                  2       /* 250MHz -  4ns */
#define K_200MHz                  3       /* 200MHz -  5ns */
#define K_100MHz                  4       /* 100MHz - 10ns */
#define K_50MHz                   5       /* 50MHz  - 20ns */
#define K_25MHz                   6       /* 25MHz  - 40ns */
#define K_20MHz                   7       /* 20MHz  - 50ns */
#define K_10MHz                   8       /* 10MHz  -100ns */
#define K_5MHz                    9       /* 5MHz   -200ns */
#define K_2_5MHz                 10       /* 2.5MHz -400ns */
#define K_2MHz                   11       /* 2MHz   -500ns */
#define K_1MHz                   12       /* 1MHz   -  1us */
#define K_500KHz                 13       /* 500KHz -  2us */
#define K_250KHz                 14       /* 250KHz -  4us */
#define K_200KHz                 15       /* 200KHz -  5us */
#define K_100KHz                 16       /* 100KHz - 10us */
#define K_50KHz                  17       /* 50KHz  - 20us */
#define K_25KHz                  18       /* 25KHz  - 40us */
#define K_20KHz                  19       /* 20KHz  - 50us */
#define K_10KHz                  20       /* 10KHz  -100us */
#define K_5KHz                   21       /* 5KHz   -200us */
#define K_2_5KHz                 22       /* 2.5KHz -400us */
#define K_2KHz                   23       /* 2KHz   -500us */
#define K_1KHz                   24       /* 1KHz   -  1ms */
#define K_500Hz                  25       /* 500Hz  -  2ms */
#define K_250Hz                  26       /* 250Hz  -  4ms */
#define K_200Hz                  27       /* 200Hz  -  5ms */
#define K_100Hz                  28       /* 100Hz  - 10ms */
#define K_Sync_R              K_Sync      /* Synchronous, rising edge  */
#define K_Sync_F                 29       /* Synchronous, falling edge */
#define K_Sync_E                 30       /* Synchronous, either edge  */
#define K_1GHz                   31       /* 1GHz   -  1ns */
```

Remarks

This sets the speed of the internal clock which samples the incoming data.

Ant8: Select a value between 100Hz and 500MHz.

Ant16: As Ant8, or the Ant16 can sample data synchronously on the rising edge of an

external clock. The maximum speed of the external clock is 100MHz.

Ant18e: As Ant16, or 1GHz, or the Ant18e can sample data synchronously on the rising edge, falling edge, or either edge of an external clock. The maximum speed of the external clock is 100MHz.

The Ants support both of the common timing sequences:

- the 5-2-1 sequence of frequencies. For instance: 50MHz, 20MHz, 10MHz
- the 1-2-4 sequence of times. For instance: 10ns, 20ns, 40ns

Example

```
RL_RESULT retval = RL_SetClockIx(h, K_100MHz );
```

```
RL_RESULT RL_GetClockIx(RL_HANDLE h, int* pClkIx);
```

Returns the current clock index to the integer pointed to by *pClkIx*.

```
RL_RESULT RL_SetTriggerPos(RL_HANDLE h, int Percent);
```

Parameters

h	handle returned from a previous RL_Initialize.
Percent	position of the trigger in the Ant's buffer, as a percentage of the buffer size.

Remarks

The trigger position divides the logic analyzer's acquisition memory into the pre-trigger part and the post-trigger part. The pre-trigger part of the memory is the minimum amount which must be filled with data samples before the logic analyzer starts searching for the trigger pattern. The post-trigger part of memory is filled after the trigger pattern is detected, then the analyzer stops.

The trigger position can vary from 1% to 99%.

Example

```
RL_RESULT retval = RL_SetTriggerPos(h, 50 );
```

```
RL_RESULT RL_GetTriggerPos(RL_HANDLE h, int* pPercent);
```

Returns the current trigger position, as a percentage, to the integer pointed to by *pPercent*.

```
RL_RESULT RL_SetThreshold(RL_HANDLE h, int ByteIx, int ThreshValX10);
```

Parameters

h	handle returned from a previous RL_Initialize.
ByteIx	must be within the byte count deduced from *Channels* returned by RL_Info().
ThreshValX10	threshold voltage in tenths of a volt.

Remarks

The threshold is the voltage level at which the analyzer discriminates between a low signal (a zero) and a high signal (a one).

The Ant8 sets the threshold to 1.4V for all channels and this function is a no-op.

The Ant16 and the Ant18e allow the threshold to be programmed to a value between 0.8V and 2.5V. For the Ant16, the threshold is set separately for 8-bit groups of channels.

ByteIx=0	Channels 0 to 7
ByteIx=1	Channels 8 to 15

For the Ant18e, the threshold is set separately for 9-bit groups of channels.

ByteIx=0	Channels 0 to 7 and 16
ByteIx=1	Channels 8 to 15 and 17

Example

```
RL_RESULT retval = RL_SetThreshold(h, 0, 12 );      // 1.2 volts
```

```
RL_RESULT RL_GetThreshold(RL_HANDLE h, int ByteIx, int*
pThreshValX10);
```

Returns the current threshold voltage, times 10, for byte *ByteIx* to the integer pointed to by *pThreshValX10*.

3.5 Starting and Stopping the Ant and Reading Status and Data

```
RL_RESULT RL_Run(RL_HANDLE h, bool run);
```

Parameters

h	handle returned from a previous RL_Initialize.
run	true: start an acquisition, false: prematurely terminate an acquisition.

Remarks

Calling this function with run=true instructs the software to configure the Ant, download all the acquisition parameters into registers, and start acquiring a data sample. The application code can then monitor the status of the Ant via RL_ GetRunStatus() - see below.

Normally the Ant will search for the trigger conditions, trigger, and stop. The acquired data can then be read back via the RL_Readback() function.

If the trigger condition does not occur the Ant will not trigger and it may be necessary to force a stop by calling RL_Run() with run=false.

Example

```
RL_RESULT retval = RL_Run(h, true );    // start
...
...
retval = RL_Run(h, false );             // stalled? force a stop
```

```
RL_RESULT RL_GetRunStatus(RL_HANDLE h, int* Status);
```

Parameters

h	handle returned from a previous RL_Initialize.
Status	pointer to an integer which will be filled in with the Ant's status.

Remarks

Status is returned as an 8-bit code:

- bit 7: set if the Ant has triggered
- bit 6: set if the Ant's acquisition memory has been written at least once
- bits 5..0: the state of the internal state machine

The state machine status in bits 5..0 is encoded as follows:

1. in the *Init* state – not started
2. in the *Prefill* state – not yet searching for a trigger
3. in the *Search* state
4. in the *Hit1* state
5. in the *Hit2* state
6. in the *Triggered* state - has triggered and is now postfilling
7. in the *Done* state - postfill completed

Note

In early iterations of the DLL, this function had a third parameter and returned the Trigger position in the acquisition memory.

Example

```
int status=0;
RL_RESULT retval = RL_GetRunStatus(h, &status );
```

```
RL_RESULT RL_Readback(RL_HANDLE h, BYTE *pBuff, int BuffSize, int
Dummy );
```

Parameters

h	handle returned from a previous RL_Initialize.
pBuff	memory to store the readback.
Buffsize	size in bytes of the space pointed to by pBuff.

Remarks

The Ant18e implements *transitional sampling*, a feature not implemented in the Ant8 and the Ant16. The Ant8 and the Ant16 report straightforward sample values - for instance, 5,5,3,7. In contrast, the Ant18e reports pairs of numbers (sample, repetition), so that (5,2), (9,3), (1,5) means that the samples were 5, 5, 9, 9, 9, 1, 1, 1, 1, 1. This is the mechanism the Ant18e

uses to get very large buffer depths – the repetition value can be up to 262144.

A byproduct of implementing transitional sampling is that the sampling buffer depth in the Ant18e varies between 6K and 8K, depending on the sampling mode used.

Readback() deals with these complexities and provides a simple, unified programming interface by implementing these mechanisms:

1. all readback data from all Ant models is reported in (sample, repetition) format. These pairs are referred to as readback data *slots*.
2. (sample, repetition) pairs are returned packed into consecutive DWORDs in the buffer pointed to by pBuff. The packing sequence is as follow:

   DWORD 0 sample 0
   DWORD 1 repetitions for sample 0
   DWORD 2 sample 1
   DWORD 3 repetitions for sample 1
   DWORD 4 sample 2
   DWORD 5 repetitions for sample 2
   ...
   ...

To read back from an Ant, the procedure is

1. determine the number of slots via RL_GetVal()
2. determine the bytes per slot via RL_GetVal(). Currently this is always 8.
3. call readback for the (number of slots) * (bytes per slot)

The results are meaningless unless the Ant is in Done state. Also RL_GetRunStatus() must have been called first.

Example

```
int status=0, trigpos=0;
RL_RESULT retval = RL_GetRunStatus(h, &status, &trigpos );
if ((status & 0x3f) == FPGA_ACQUISITION_STATE_done) {
  int TotalSlots   = (int)RL_GetVal(h, RL_GETVAL_SLOTCOUNT);
  int BytesPerSlot = (int)RL_GetVal(h, RL_GETVAL_SLOTBYTES);

  int ByteCount = TotalSlots * BytesPerSlot;

  BYTE *Buff = new BYTE[ByteCount];
  RL_RESULT retval = RL_Readback(h, Buff, ByteCount, 0 );
...
}
```

Note

In early iterations of the DLL the Dummy fourth parameter to this function set the first sample to be read back, which is now always sample 0.

3.6 Frequency Counter Functions

The frequency counter functions only apply to the Ant18e. See application note AN-7 *The Ant18e Frequency and Event Counter* for more information.

The frequency counting functionality is supported by these software library functions:

- RL_FCRun sets the parameters and starts or stops the frequency counter.
- RL_FCGetStatus returns the frequency counter status.
- RL_FCGetCount returns the per-channel status and the per-channel counts.

```
RL_RESULT RL_FCRun(RL_HANDLE h, int AOp, int AEdges, int AGateLength,
char *APat, bool ARun)
```

Parameters

h	handle returned from a previous RL_Initialize.
AOp	operation type. FC_OP_FREQ or FC_OP_PERIOD or FC_OP_EVENT.
AEdges	define the start and stop edges in period mode.
AGateLength	define the gate length in frequency counter mode.
Apat	pointer to the pattern for event counting.
ARun	true (1) to start, false (0) to stop.

Remarks

The parameter values for AOp, AEdges, and AGateLength are specified in antif.h

The pattern is a string of high ('1'), low ('0'), and don't care ('-') characters. Where necessary it is padded with don't care characters. The first character in the string applies to channel 0, the second to channel 1, and so on.

To force a counter stop, set ARun to 0 - the other parameters are *don't care* in this case.

In period mode, the counter measures from the A edge to the B edge. The constants for rising and falling A and B edges are OR'd together to form the AEdge value.

Example

```
RL_RESULT retval = RL_FCRun(h, FC_OP_PERIOD,
 FC_EDGEA_RISING | FC_EDGEB_RISING,
 0, "--------", true);
// counter now running in period measurement mode
```

```
RL_DLL RL_FCGetStatus(RL_HANDLE h, int *pStat, int Alen)
```

Parameters

h	handle returned from a previous RL_Initialize.
pStat	pointer to the buffer to receive the Frequency Counter status.
Alen	number of channels to report, typically 19 for the Ant18e.

Remarks

The first value (pStat[0]) has bit zero set if the frequency counter is busy, clear if idle. Other bits are reserved.

The per-channel status is encoded as follows (other bits are reserved) in the remaining ints – pStat[1] to pStat[18]:

```
bits 2..0   0      channel is idle
            1      channel is counting
            2      channel is finished
            3      reserved
            4      channel was forced to stop by host command (forcedFinished)
            5 to 7 reserved
bit 3       set if the A edge has been seen
bit 4       set if the B edge has been seen
```

Example

```
int status[1+18];
RL_RESULT retval = RL_FCGetStatus(h, status, 19);
bool CounterIsBusy = ((status[0] & 1) == 1);
```

```
RL_RESULT RL_FCGetCounts(RL_HANDLE h, __int64 *pVal, int AChans);
```

Parameters

h	handle returned from a previous RL_Initialize.
pVal	pointer to the buffer to receive the per-channel counter values.
AChans	number of channels to read, starting from channel 0. Usually 18.

Remarks

The per-channel counter values are 64-bit unsigned integers. The current count value can be read back wile the counter is running.

Example

```
__int64 val[18];
RL_RESULT retval = RL_FCGetCount(h, val, 18);
```

3.7 Miscellaneous Functions

```
RL_RESULT RL_GetPinStatus(RL_HANDLE h, char* pStatus, int StatusLen);
```

Parameters

h	handle returned from a previous RL_Initialize.
pStatus	memory to store the status. One character per channel.
StatusLen	size in chars of the space pointed to by pStatus.

Remarks

This function halts any acquisition under way, reconfigures the Ant, if necessary, and samples the input channels. The status returned is encoded as a character string. *R*, *F*, and *B* denote that a rising edge, a falling edge, or both edges have been seen since the last call to this function. The status string is not automatically null terminated.

Character	*Input signal*
0	low
1	high
R	rising
F	falling
B	rising and falling

Example

```
char Stat[17];
memset(Stat, 0, sizeof(Stat));
RL_RESULT retval = RL_GetPinStatus(h, Stat, sizeof(Stat) );
```

```
RL_RESULT RL_GetLastErrorMessage(RL_HANDLE h, char* s, int len);
```

Parameters

h	handle returned from a previous RL_Initialize.
s	buffer to store the error message.
len	size in chars of the space pointed to by s.

Remarks

This function returns the last error message from the server as a null terminated character string. The message is truncated if the buffer is too small.

Example

```
char Err[200];
memset(Err, 0, sizeof(Err));
RL_RESULT retval = RL_GetLastErrorMessage(h, Err, sizeof(Err) );
printf("Last error message was: %s\n", Err);
```

```
RL_RESULT RL_Heartbeat(RL_HANDLE h);
```

Parameters

h	handle returned from a previous RL_Initialize.

Remarks

How does a server distinguish between a quiet client and a client who has died? Usually via a *heartbeat* message from the client. If the heartbeat is not heard for a certain period – typically ten minutes – the server can assume the client has died and any resources the client was holding (for instance an Ant which has been opened) can be released for another client application to use. Although not implemented in the current release of the server, this functionality is implemented via the RL_Heartbeat function, which should be called every minute of so.

Example

```
RL_RESULT retval = RL_Heartbeat(h);
```

```
RL_RESULT RL_SetScratchReg(RL_HANDLE h, int local, int val);
```

Parameters

h	handle returned from a previous RL_Initialize.
local	zero: store in the global register, non-zero: store in this client's register
val	value stored in the server.

Remarks

This function stores an 32 bits of information in a register in the server. Each client has access to two registers:

- the private register can only be written or read by the client
- the global register can be written or read by any connected client.

Example

```
RL_RESULT retval = RL_SetScratchReg(h, 0, 1); // global reg
```

```
RL_RESULT RL_SetVal(RL_HANDLE h, int param, QWORD val);
```

Parameters

h	handle returned from a previous RL_Initialize.
param	index of the parameter to set. None defined at present.
val	parameter value.

Remarks

This is an all-purpose function for setting parameters. The parameter is an unsigned 64 bit integer.

```
RL_RESULT RL_GetVal(RL_HANDLE h, int param, QWORD* pVal);
```

Parameters

h	handle returned from a previous RL_Initialize.
param	index of the parameter to get. None defined at present.
pVal	pointer to location to store the parameter.

Remarks

This is an all-purpose function for getting parameters. The parameter is returned as an unsigned 64 bit integer.

The following parameters are defined:

Parameter	Meaning
RL_GETVAL_SAMPLECOUNT	Total samples, summed by adding up the repetitions for each slot. For the Ant18e, this number can be several billion.
RL_GETVAL_TRIGGERPOS	Trigger position, a Sample value, not a Slot value.
RL_GETVAL_SLOTCOUNT	Slot Count. A small integer.
RL_GETVAL_SLOTBYTES	Bytes per slot for a (value, count) pair. See under RL_Readback().

```
RL_RESULT RL_ClockInfo(int ClkIx, int *ClockPeriod, int
            *ClockPeriodUnit, char *pTagBuff, int TagBuffSize)
```

Parameters

ClkIx	clock index. See the table under RL_SetClockIx().
ClockPeriod	pointer to an integer which is filled in with the clock period.
ClockPeriodUnit	pointer to an integer which is filled in with the clock period units.
pTagBuff	pointer to an character buffer which is filled in with the clock name.
TagBuffSize	length of the character buffer

Remarks

The clock period units codes are defined in antif.h

Calling this function with a clock index of K_100MHz (i.e. 4) will return ClockPeriod=10 and ClockPeriodUnit=TIME_UNIT_NS, and the tag buffer will be filled in with " 100MHz".

4. Default Acquisition Parameters

A successful call to RL_OpenAnt() sets the acquisition parameters to default values, as follows:

Parameter	*Value*
hit patterns	all “--------...” and AndCombine=true
X functions	all “(P0)”
state machine functions	all “F” except the transition from Search state to Triggered state = “(I0)”
sample clock	100MHz
trigger position	50%
timer/counter	2 and the timer/counter is a counter
thresholds	all 1.4V

These values can be confirmed by calling the relevant RL_Get...() functions.

5. Revision History

Date	*Revision*
25 January 2006	Initial Version.
21 February 2007	New functions for Frequency Counting. General updating.

For more information see www.rockylogic.com