United Electronic Industries

The High-Performance Alternative

DNA/DNR-IRIG-650

—

User Manual

IRIG-A, B, E and G Timing Generation and Synchronization board for the PowerDNA Cube and PowerDNR RACKtangle

Release 4.6

March 2019

PN Man-DNx-IRIG-650-319

Information furnished in this manual is believed to be accurate and reliable. However, no responsibility is assumed for its use, or for any infringement of patents or other rights of third parties that may result from its use.

All product names listed are trademarks or trade names of their respective companies.

See the UEI website for complete terms and conditions of sale:
http://www.ueidaq.com/cms/terms-and-conditions/

CE

Contacting United Electronic Industries

Mailing Address:

27 Renmar Avenue
Walpole, MA 02081
U.S.A.

For a list of our distributors and partners in the US and around the world, please contact a member of our support team:

Support:

Telephone: (508) 921-4600
Fax: (508) 668-2350

Also see the FAQs and online “Live Help” feature on our web site.

Internet Support:

Support: support@ueidaq.com
Website: www.ueidaq.com
FTP Site: ftp://ftp.ueidaq.com

Product Disclaimer:

WARNING!

DO NOT USE PRODUCTS SOLD BY UNITED ELECTRONIC INDUSTRIES, INC. AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS.

Products sold by United Electronic Industries, Inc. are not authorized for use as critical components in life support devices or systems. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness. Any attempt to purchase any United Electronic Industries, Inc. product for that purpose is null and void and United Electronic Industries Inc. accepts no liability whatsoever in contract, tort, or otherwise whether or not resulting from our or our employees' negligence or failure to detect an improper purchase.

Specifications in this document are subject to change without notice. Check with UEI for current status.

Table of Contents

List of Figures

Chapter 1 Introduction

This document outlines the feature set and use of the DNR- and DNA-IRIG-650 layer. The IRIG-650 Timing Generation and Synchronization board is a module for the UEI PowerDNA I/O Cube (DNA-IRIG-650) and the DNR-1G HalfRACK and RACKtangle chassis (DNR-IRIG-650). The DNR version is electronically identical to the DNA version except that the DNR version is designed to plug into a RACKtangle backplane instead of a Cube chassis.

This module may be used to capture Inter-range Instrumentation Group (IRIG) data when the Cube or RACKtangle is slaved to an external master timing device. It also provides IRIG outputs that allow the Cube or RACKtangle to serve as master time keeper for the system.

The DNx-IRIG-650 provides inputs for standard analog, modulated IRIG signals as well as non-modulated DC (DCLS or NRZ) and Manchester II inputs. In addition to the IRIG inputs, the board also allows the user to provide an external 10 MHz master clock and/or a 1 PPS synchronization pulse. A generic digital input may also be used to directly capture event timing.

The DNx-IRIG-650 can also be configured as an IRIG source which will provide timing and synchronization for other devices in the system. The board provides both modulated analog and digital IRIG outputs as well as 10 MHz and 1 PPS synchronization and timing (DCLS/Manchester II) signals.

The boards also include a built-in GPS interface. UEI recommends the use of high gain, active GPS antennas, such as the Symmetricom AT575-142 or equivalent.

1.1 Organization of Manual

This DNx IRIG-650 User Manual is organized as follows:

- **Chapter 1 Introduction**
  This chapter provides an overview of DNx-IRIG-650 Timing Board features, device architecture, connectivity, and logic. This chapter also describes various inputs and outputs.
- **Chapter 2 Programming with the High-Level API**
  This chapter provides an overview of the how to use Framework to create a session, configure the session for the various types of inputs and outputs, and how to interpret results.
- **Chapter 3 Programming with the Low-Level API**
  This chapter refers the reader to a Reference API that defines low-level functions and commands for configuring and using the IRIG-650 series layer.
- **Appendix A Accessories**
  This appendix provides a list of accessories available for the board.
- **Index**
  This is an alphabetical listing of the topics covered in this manual.

Manual Conventions

To help you get the most out of this manual and our products, please note that we use the following conventions:

Tips are designed to highlight quick ways to get the job done or to reveal good ideas you might not discover on your own.

NOTE: Notes alert you to important information.

CAUTION! *Caution advises you of precautions to take to avoid injury, data loss, and damage to your boards or a system crash.*

Text formatted in **bold** typeface generally represents text that should be entered verbatim. For instance, it can represent a command, as in the following example: “You can instruct users how to run setup using a command such as **setup.exe**.”

Text formatted in `fixed` typeface generally represents source code or other text that should be entered verbatim into the source code, initialization, or other file.

Examples of Manual Conventions

Before plugging any I/O connector into the Cube or RACKtangle, be sure to remove power from all field wiring. Failure to do so may cause severe damage to the equipment.

Usage of Terms

Throughout this manual, the term “Cube” refers to either a PowerDNA Cube product or to a PowerDNR RACKtangle™ rack mounted system, whichever is applicable. The term DNR is a specific reference to the RACKtangle, DNA to the PowerDNA I/O Cube, and DNx to refer to both.

1.2 Layer Features

The DNx-IRIG-650 layer has the following features:

- IRIG-A, B, E and G output allows IRIG-650 to provide timing signals
- IRIG-A, B, E and G input allows IRIG-650 to synchronize with external IRIG-A, B, E and G sources
- Modulated or DC level inputs and outputs
- 1 PPS output
- Event input captures timing to UTC (Coordinated Universal Time)
- Direct GPS input (active antenna)
- 10 MHz, 1 ppm time base or slaved to external 10 MHz

Protocols Supported:

IRIG-A: A00x, ABx

IRIG-B: B00x, B12x,B IEEE 1344, B TrueTime

IRIG-E: E00x, E11x, E12x

IRIG-G: G00x, G14x

Time Source:

The IRIG-650's time keeper can be synchronized with the following sources:

- IRIG AM-modulated input timecode
- IRIG DCLS (i.e. NRZ-L code) or Manchester II code
- GPS signal (sync'd with GPS 1 PPS clock, time derived from GPRMC string)
- Precise external 1PPS (local time should be set by the user)

Inputs:

- AM (timecode), AMIn, DCLS, MII (Manchester II)
- GPS RFIn
- 1 PPS (TTL0, TTL1 Line)
- External 10 MHz (TTL0)
- External events (TTL-1u or SYNCx lines)

Outputs:

- Time Code: AM (AMOut), DCLS (TTL-OutX), Manchester II
- 10 PPS, 100 PPS (TTL-OutX, SYNCx)
- 1MHz, 5 MHz, 10 MHz (TTL-OutX, SYNCx)
- Custom PLL frequency (1Hz -1 MHz, 4 digits accurate)
- 1 PPS, 1 PPM, 1PPM (TTL-OutX, SYNCx)

1.3 Technical Specification

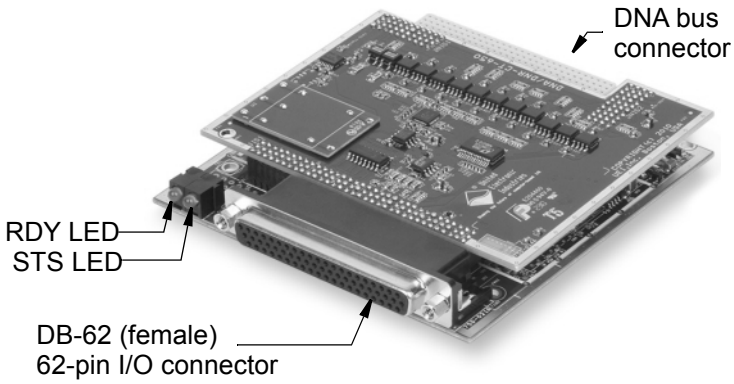
The technical specifications of the DNA/DNR-IRIG-650 IRIG Timing Layer.

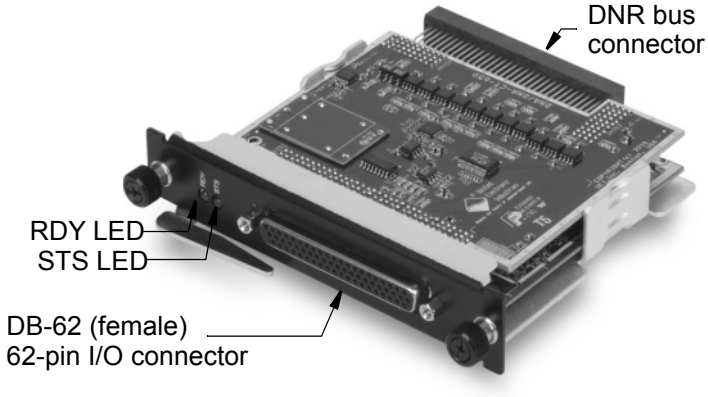
Inputs	
IRIG Analog inputs	A, B, E and G types supported
Modulation ratio	3:1 to 6:1
Input amplitude	500 mV to 5 V P-P (AC coupled)
Input impedance	> 10 k Ω
IRIG-B DC inputs	3.3/5 V logic compliant
10 MHz input	3.3/5 V logic compliant 40% to 60% duty cycle
1 PPS input	3.3 and 5 V logic compliant
GPS	
Antenna configuration	Only active antennas are supported
Position (Velocity) accuracy	1.8 m (0.1 m/S) rms
UTC time accuracy	± 50 nS rms
Outputs	
IRIG Output types	A, B, E and G types supported
Analog output	3:1 ratio, 4 V P-P output (50 ohm)
Digital output high voltage	1.1 V - 50 Ω (min) 2.4V - 1 Meg Ω (min)
Digital output low voltage	0.3 V - 50 Ω (max) 0.7V - 1 Meg Ω (max)
Sync and Clock outputs	TTL/CMOS compatible
Output timing signal selection	Std 1 & 10 PPS/PPM plus custom
Output clock selection	1, 5 and 10 MHz plus custom freqs.
On-Board Clock	
Frequency	10 MHz
Initial accuracy	50 PPB
Temperature stability	50 PPB over full temp range
Time stability	300 PPB per year
Output Voltage	TTL/CMOS compatible
General	
Power consumption	2W
Operating range	Tested -40 to +85 °C
Isolation	350 Vrms between all IRIG signals and the chassis. (GPS is not isolated)
Humidity range	0-95%, non-condensing
Vibration *IEC 60068-2-6* *IEC 60068-2-64*	5 g, 10-500 Hz, sinusoidal 5 g (rms), 10-500Hz, broad-band random
Shock *IEC 60068-2-27*	50 g, 3 ms half sine, 18 shocks @ 6 orientations 30 g, 11 ms half sine, 18 shocks @ 6 orientations
Altitude	to 70,000 feet

Table 1-1. Technical Specifications

1.4 Indicators

Indicators of the layers are labelled in the pictures below:





1.5 Simplified Block Diagram

The figure below shows a simplified block diagram of the layer's architecture:

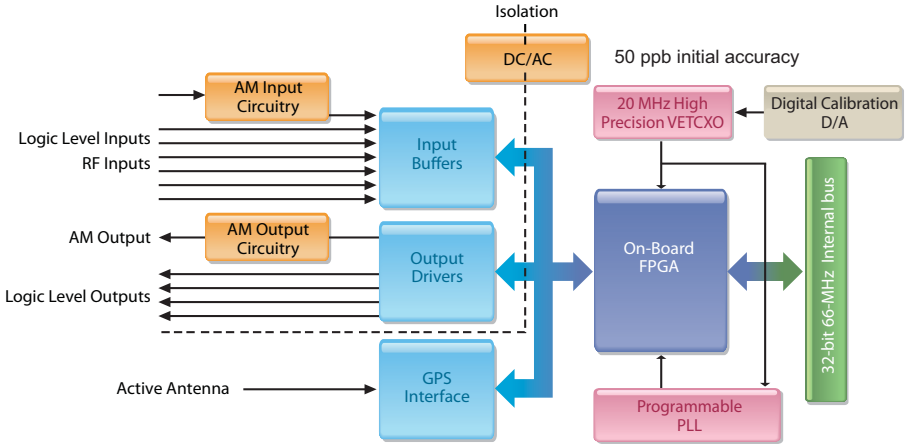


Figure 1-1. Simplified Block Diagram of the IRIG-650

1.6 Functional Description

The following is a functional block diagram of The DNx-IRIG-650 IRIG Timing Generation and Synchronization Board.

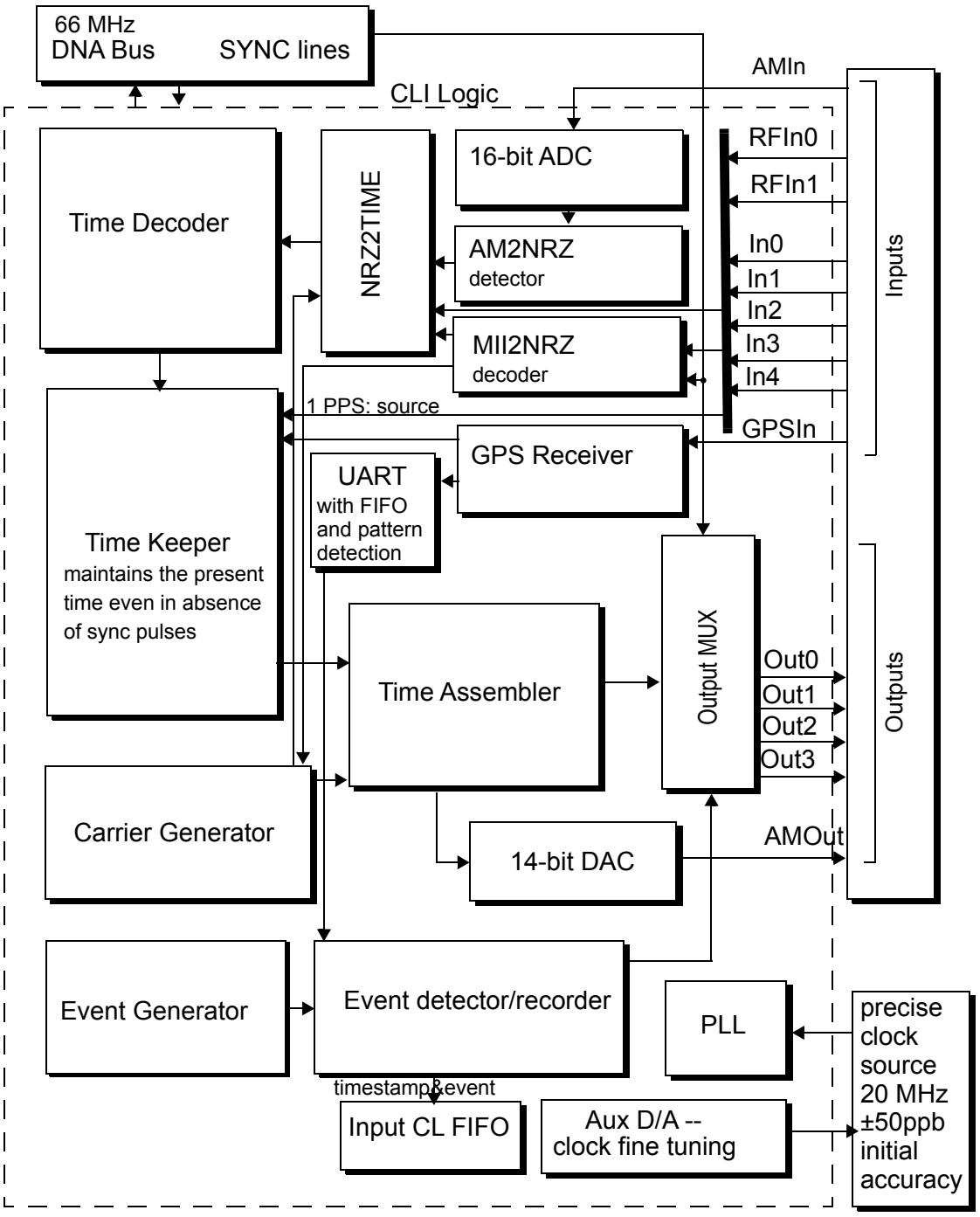


Figure 1-2. Functional Diagram of DNx-IRIG-650 board

The IRIG input time code can be received from one of the following sources:

- AM-modulated signal acquired by the 16-bit ADC and processed by the AM2NRZ module. This module automatically detects the zero crossing, adjusts the zero level, extracts the high- (mark) and low- (space) amplitude ratio and generates the resulting NRZ-L (i.e. DCLS) code.
- Manchester II encoded signal processed by the MII2NRZ module. You must select the correct carrier for the NRZ data extraction to work.
- Direct NRZ input; where the input signal is direct-current level shifted (DCLS) signal which is pulse-width coded but is not AM-modulated.
- A forced initial time setup from the DNA host, immediate, and on 1PPS edges

NOTE: *The encoding "non-return to zero, level (NRZ-L)" is synonymous with "direct-current level shift (DCLS)" encoded signals*

A 1PP*x* (e.g. 1PPS) input "on-time pulse" can be sourced from NRZ time code. If an NRZ-encoded time code is not available, an external 1PPS signal can be delivered from a PowerDNx SYNC line, an external input, or the GPS input.

NRZ information is down-converted by the NRZ2TIME module to create time characters: "Px" (position identifier), "0" (logic 0), "1" (logic 1), plus idle characters. In most time codes, the IDLE character is not used, and if present, it usually indicates a lost carrier signal.

After the input data stream is parsed into time characters, the characters are processed by the Time Decoder module to: extract an "on-time" pulse, copy incoming data into double-buffered time messages, validate these messages using a validation table. Each validated message is further processed to extract time information, which is then copied into timing registers.

The current time is maintained in the Time Keeper module, which receives time from the Time Decoder and keeps track of the time and/or date increments. If the current time does not match the time received from the decoder, an interrupt is generated. When this occurs, the time received from the decoder is then accepted as the valid current time.

In hardware the Time Keeper module always maintains a time that is 1 second ahead of the time just received - but the software does correct the returned time. The reason for this is that the time code received from the Time Assembler module is known to be about 1 time frame behind its own "on-time" pulse.

If an external time source is not available, the Time Keeper module maintains time based on internal or external 1 PPS pulses and the initial time set by the host, or based on the last valid time recovered from the time source. The 1PPS time interval that is used for the flywheel counter should pass strict validation prior to use, and in case of lost or invalid input data, requires at least two periods of valid 1 PPS signal prior to use. By default, 1 PPS valid interval is set to ±50µs. The flywheel takes in the 100MHz clock and synchronizes to it on 1PPS pulse. The 100MHz base clock of the IRIG-650 is generated by a PLL on the FPGA that multiplies the output of a very high precision 20MHz voltage controlled, temperature compensated oscillator with ±50ppb frequency stability and maximum drift of ±500ppb, the majority of which will happen within the first year of operation and can be compensated by the in-factory calibration which uses a Rubidium Standard source.

The phase noise specification of the 20MHz oscillator is:

PHASE NOISE:10Hz OFFSET: -105dBc/Hz

100Hz OFFSET: -125dBc/Hz

1kHz OFFSET: -135dBc/Hz

10kHz OFFSET: -145dBc/Hz

The Time Keeper module can use either raw binary seconds or binary-coded decimal (BCD) seconds/minutes/hours from the input code to read the current time.

The Time Assembler module accepts the current time from the Time Keeper and uses it to create an output time code with Manchester II, AM, and DCLS coding. (Note that the output format does not have to be the same format as an input.)

In addition to the time keeping & generating functionality described above, the IRIG-650 also provides event recording and event generating functions.

Event recording allows you to timestamp and store locations of up to four events from the following event sources:

- SYNC bus lines of the Cube or RACKtangle
- External inputs
- Errors (lost synchronization inputs, invalid 1 PPS, etc.)
- Start/stop trigger broadcast commands

The above recorded events may be streamed into the output FIFO, which allows you to synchronize data acquired by any other DNR/DNA boards with time that is accurately kept by the IRIG-650 layer. The above recorded events can also be routed to TTL or SYNCx lines.

Event generation allows creation of one-time or repeatable events based on the precise time that is maintained by the IRIG-650 layer. The maximum recommended rate of events is 100 kHz and the maximum resolution is 10 nsec.

Event generation includes a digital PLL mode that generates a configurable number of pulses per period (2MHz max), designed to adjust its period to a 1PPS source from the TimeKeeper or from a TTL input.

1.7 Layer Connectors and Wiring

The figure below illustrates the pinout of the IRIG-650.

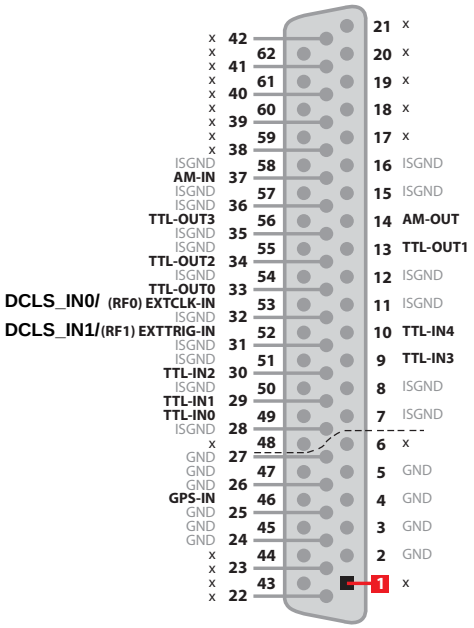


Figure 1-3. Pinout for the DNx-IRIG-650 series layer

Capable of attaching to the 62-pin connector are the plug-in break-out board or the DNA-CBL-650 cable, which is listed in the appendix. The following signals are brought out to the female BNC connectors on the DNA-CBL-650:

- AM-IN: IRIG AM Input (pin 37)
- AM-OUT: IRIG AM Output (pin 14)
- DCLS-IN: ExtClk-Input (pin 53)
- GPS-IN: GPS antenna input (pin 46)

The remaining signals are brought out to a 37-pin, Female connector at the end of the DNA-CBL-650 as shown in the appendix. This cable may be plugged into a DNA-STP-37 screw terminal panel or other panel. Please note that 12 of these signals are twisted pairs with their respective grounds as shown in the appendix.

Chapter 2 Programming with the High Level API

This section describes how to control the DNx-IRIG-650 using the UeiDaq Framework High Level API.

UeiDaq Framework is object oriented and its objects can be manipulated in the same manner from different development environments such as Visual C++, Visual Basic or LabVIEW.

The following section focuses on the C++ API, but the concept is the same no matter what programming language you use.

Please refer to the “UeiDaq Framework User Manual” for more information on use of other programming languages.

2.1 Creating a Session

The Session object controls all operations on your PowerDNA device. Therefore, the first task is to create a session object:

```
// create a session object

CUeiSession irigSession;
```

2.2 Configuring the Resource String

UeiDaq Framework uses resource strings to select which device, subsystem and channels to use within a session. The resource string syntax is similar to a web URL:

```
<device class>://<IP address>/<Device Id>/<Subsystem><Channel list>
```

For PowerDNA and RACKtangle, the device class is **pdna**.

The IRIG-650 is programmed using the subsystem **irig** (letter-case insensitive).

For example, the following resource string selects IRIG input line 0 on device 1 at IP address 192.168.100.2: “pdna://192.168.100.2/Dev1/Irig0”

2.2.1 Configuring Time Keeper Input

Use the Session object’s method “`CreateIRIGTimeKeeperChannel`” to configure the time keeper channel and parameters associated with the channel.

The following sample code shows how to configure the time keeper channel of a IRIG-650 set as device 1:

```
// Configure the time keeper

CUeiIRIGTimeKeeperChannel* pTKChannel =
               irigSession.CreateIRIGTimeKeeperChannel(
                           "pdna://192.168.100.2/Dev1/Irig0",
                           UeiIRIG1PPSInternal,
                           autoFollow);
```

It configures the following parameters:

- **1PPS source:**The 1PPS signal source, that can be any of the following values:

1PPS Source Value	Description
`UeiIRIG1PPSInternal`	1PPS signal is generated internally with precision oscillator
`UeiIRIG1PPSInputTimeCode`	1PPS signal is derived from input timecode
`UeiIRIG1PPSGPS`	1PPS signal is derived from GPS
`UeiIRIG1PPSRFIn`	1PPS signal is derived from signal connected on RF input
`UeiIRIG1PPSExternalTTL0`	External 1PPS signal is connected to TTL0 input pin
`UeiIRIG1PPSExternalTTL1`	External 1PPS signal is connected to TTL1 input pin
`UeiIRIG1PPSExternalTTL2`	External 1PPS signal is connected to TTL2 input pin
`UeiIRIG1PPSExternalTTL3`	External 1PPS signal is connected to TTL3 input pin
`UeiIRIG1PPSExternalSync0`	External 1PPS signal is connected to SYNC0 bus line
`UeiIRIG1PPSExternalSync1`	External 1PPS signal is connected to SYNC1 bus line
`UeiIRIG1PPSExternalSync2`	External 1PPS signal is connected to SYNC2 bus line
`UeiIRIG1PPSExternalSync3`	External 1PPS signal is connected to SYNC3 bus line

- **Auto-follow:** If selected, external 1PPS source does not deliver pulses (because of a break in timecode transmission, for example). The Timekeeper can switch to internal timebase when externally derived one is not available.

In addition you can set additional parameters using the channel object methods (under LabVIEW use property node):

- **Nominal Value enabled:** Select whether to use nominal period (i.e. 100E6 pulses of 100MHz base clock) or the period measured by timekeeper (it measures and averages number of base clock cycles between externally derived 1PPS pulses when they are valid).

```
// enable nominal value

pTKChannel->EnableNominalValue(true);
```

- **Sub PPS enabled:** Select whether external timebase is slower than 1PPS or is not derived from the timecode.

```
// Disable sub PPS

pTKChannel->EnableSubPPS(false);
```

- **Initial time:** The initial time loaded in time keeper.

```
// Initial Time

tUeiANSITime now;…
pTKChannel->SetInitialTime(now)
```

2.2.2 IRIG Output

Use the method `CreateIRIGOutputChannel()` to configure the time code output on your IRIG-650.

```
// configure the time code output

CUeiIRIGOutputChannel* pOutChannel =
           irigSession.CreateIRIGOutputChannel(
                       "pdna://192.168.100.2/Dev1/Irig0",
                       timeCodeformat);
```

It configures the following parameters:

- **Timecode Format:** the format used to generate the time code.
  - `UeiIRIGTimeCodeFormatA`: IRIG-A
  - `UeiIRIGTimeCodeFormatB`: IRIG-B
  - `UeiIRIGTimeCodeFormatE_100Hz`: IRIG-E 100Hz
  - `UeiIRIGTimeCodeFormatE_1000Hz`: IRIG-E 1000Hz
  - `UeiIRIGTimeCodeFormatG`: IRIG-G

In addition you can set the following parameter using the channel object methods (under LabVIEW use property node):

- **Start when input is valid:** If selected, the output time coder waits for the input time decoder to receive a valid time code before starting.

```
// start when input is valid

pOutChan->EnableStartWhenInputValid(true);
```

2.2.3 IRIG Input

Use the method `CreateIRIGInputChannel()` to configure the time code input on your IRIG-650.

```
// configure the time code input

CUeiIRIGInputChannel* pInChannel =
          irigSession.CreateIRIGOutputChannel(
                      "pdna://192.168.100.2/Dev1/Irig0",
                      decoderInput,
                      timeCodeformat);
```

It configures the following parameters:

- **Decoder Input Type:** The source of the incoming timecode
  - `UeiIRIGDecoderInputAM`: Time code is provided as an AM signal
  - `UeiIRIGDecoderInputManchesterRF0`: Time code is provided as a Manchester II code on RF input 0
  - `UeiIRIGDecoderInputManchesterRF1`: Time code is provided as a Manchester II code on RF input 1
  - `UeiIRIGDecoderInputManchesterIO0`: Time code is provided as a Manchester II code on I/O input 0
  - `UeiIRIGDecoderInputManchesterIO1`: Time code is provided as a Manchester II code on I/O input 1
  - `UeiIRIGDecoderInputNRZRF0`: Time code is provided as a NRZ code on RF input 0
  - `UeiIRIGDecoderInputNRZRF1`: Time code is provided as a NRZ code on RF input 1
  - `UeiIRIGDecoderInputNRZIO0`: Time code is provided as a NRZ code on I/O input 0
  - `UeiIRIGDecoderInputNRZIO1`: Time code is provided as a NRZ code on I/O input 1
  - `UeiIRIGDecoderInputGPS` : Time code is taken from GPS NMEA message
- **Timecode Format:** the format used to generate the time code.

In addition you can set the following parameters using the channel object methods (under LabVIEW use property node):

- **Idle character:** Determines whether idle character in the timing byte stream are accepted.

```
// disable idle character

pInChan->EnableIdleCharacter(false);
```

- **Single P0 Marker:** Determines whether to use only one marker P0 in the timing byte stream.

```
// Enable single P0 marker

pInChan->EnableSingleP0Marker(true);
```

2.2.4 GPS Input

Use the method `Read()` for CUeiIRIGReader to read the NMEA string from the GPS receiver associated with the IRIG port.

2.2.5 Driving the TTL outputs

Use the method `CreateIRIGDOTTLChannel()` to configure the TTL outputs on your IRIG-650. The 4 TTL outputs are represented using only one channel object.

```
// configure TTL outputs

CUeiIRIGDOTTLChannel* pTTLChan =
          irigSession.CreateIRIGDOTTLChannel(
                      "pdna://192.168.100.2/Dev1/Irig0",
                      Line0Source,
                      Line1Source
                      Line2Source
                      Line3Source);
```

It configures the following parameters:

- **TTL output 0 source:** the source used to generate the TTL pattern out of line 0.
  - `UeiIRIGDOTTLAMtoNRZ`:AM->NRZ output
  - `UeiIRIGDOTTLGPSFixValid`: GPS Fix Valid output
  - `UeiIRIGDOTTLGPSAntennaShorted`: GPS Antenna Shorted output
  - `UeiIRIGDOTTLGPS_AntennaOK`: GPS Antenna Ok output
  - `UeiIRIGDOTTLGPSTxD1`: GPS TXD1 (COM1) output
  - `UeiIRIGDOTTLGPSTxD0`: GPS TXD0 (COM0) output
  - `UeiIRIGDOTTLManchesterIICarrier`: Recovered Manchester II carrier
  - `UeiIRIGDOTTLManchesterIItoNRZ`: Decoded Manchester II -> NRZ sequence
  - `UeiIRIGDOTTLSYNC3`: Drive output from sync[3]
  - `UeiIRIGDOTTLSYNC2`: Drive output from sync[2]
  - `UeiIRIGDOTTLSYNC1`: Drive output from sync[1]
  - `UeiIRIGDOTTLSYNC0`: Drive output from sync[0]
  - `UeiIRIGDOTTLOutputCarrierFrequency`: Output carrier frequency
  - `UeiIRIGDOTTLPLLFrequency`: Custom frequency output (from PLL)
  - `UeiIRIGDOTTLPrecision10MHZ`: Precision 10MHz
  - `UeiIRIGDOTTLPrecision5MHZ`: Precision 5MHz
  - `UeiIRIGDOTTLPrecision1MHZ`: Precision 1MHz
  - `UeiIRIGDOTTLNRZStartStrobe`: Output NRZ start strobe
  - `UeiIRIGDOTTLManchesterIITimeCode`: Manchester II output time code
  - `UeiIRIGDOTTLNRZTimeCode`: NRZ output time code

  - UeiIRIGDOTTLGPS1PPS: Re-route GPS 1PPS pulse
  - UeiIRIGDOTTL1PPH: 1PPH pulse
  - UeiIRIGDOTTL1PPM: 1PPM pulse
  - UeiIRIGDOTTL1PPS: 1PPS pulse
  - UeiIRIGDOTTL0_1S: 0.1sec pulse
  - UeiIRIGDOTTL0_01S: 0.01sec pulse
  - UeiIRIGDOTTLLogic1: Drive output with 1
  - UeiIRIGDOTTLLogic0: Drive output with 0

- **TTL output 1 source:** the source used to generate the TTL pattern out of line 1
- **TTL output 2 source:** the source used to generate the TTL pattern out of line 2
- **TTL output 3 source:** the source used to generate the TTL pattern out of line 3

In addition, you can set the following parameters using the channel object methods (under LabVIEW use property node):

- **40 ns pulse:** Set pulse width to 40ns instead of the default 60µs.

```
// enable 40 ns pulses on TTL line 1
pTTLChan->Enable40nsPulse(1, true)
```

- **Use one or two TTL drivers:** Enables the second TTL driver (provides stronger driving capabilities and sharper edges).

```
// enable dual TTL driver on all outputs
pTTLChan->EnableTwoTTLBuffers(true);
```

- **Drive Sync line:** Drive sync line instead of TTL output line.

```
// configure line 3 to drive sync line 3
pTTLChan->DriveSyncLine(3, true);
```

2.3 Configuring the timing

You can only configure the IRIG-650 to run in simple mode (point by point).

In simple mode, the delay between reads is determined by software on the host computer.

The following sample shows how to configure the simple mode.

```
// configure timing
irigSsession.ConfigureTimingForSimpleIO();
```

2.4 Reading data

Reading current time data from the IRIG-650 is done using a `reader` object.

The following sample code shows how to create an IRIG reader object and read samples.

```
// create a reader and link it to the IRIG session's stream

CUeiIrigReade reader(irigSession.GetDataStream());
```

```
// read current time in BCD format

tUeiBCDTime bcdtime;
reader.Read(&bcdtime);
```

```
// read current time in SBS (straight binary seconds) format

tUeiSBSTime sbstime;
reader.Read(&sbstime);
```

```
// read current time in ANSI C format;

tUeiANSITime ansitime;
reader.Read(&ansitime);
```

2.5 Cleaning-up the Session

The session object will clean itself up when it goes out of scope or when it is destroyed. To reuse the object with a different set of channels or parameters, you can manually clean up the session as follows:

```
// clean up the session

irigSession.CleanUp();
```

Chapter 3 Programming with the Low Level API

The low-level API offers direct access to PowerDNA DAQBios protocol and allows you to directly access device registers.

Where possible, we recommend that you use the UeiDaq Framework (*see Chapter 2*), which is easier to use.

You should need to use the low-level API only if you are using an operating system other than Windows.

Please refer to the `API Reference Manual` document under:

Start » Programs » UEI » PowerDNA » Documentation

for pre-defined types, error codes, and functions for use with this layer.

The application developer is encouraged to first explore the existing source code examples. The sample code provided with the Software Suite is self-documented and the application developer will find it a good starting point.

Section 3.2 of this manual provides a set of useful programming techniques intended to help you to reshape the sample code to fit your application quickly. The code samples are self-explanatory, however this section elaborates the more complicated details with step-by-step descriptions that provide insight into the behavior behind the examples.

3.1 Low-level Functions

Refer to the API Reference Manual for detailed descriptions of the following low-level functions of IRIG-650:

Function	Description
DqAdv650AssignTTLOutputs	Sets the mode for the four TTL output lines.
DqAdv650ClockCalibration	Read/write/enable/disable the calibration counter.
DqAdv650ConfigEvents	Configures asynchronous events to be sent from the layer.
DqAdv650ConfigTimekeeper	Configures the timekeeper mode of operation.
DqAdv650Enable	This function enables/disables configured subsystems.
DqAdv650EnableGPSTracking	Enable GPS time tracking functionality.
DqAdv650GetEventStatus	Reads the event status along with event capture registers.
DqAdv650GetGPSStatus	Reads the <status>, <date> and <time> word from GPS.
DqAdv650GetTimeSBS, DqCmd650GetTimeSBS	Latches and returns time captured at the moment of latching it in binary format.
DqAdv650GetTimeBCD, DqCmd650GetTimeBCD	Latches and returns time captured at the moment of latching it in BCD format.
DqAdv650GetInputTimecode	Retrieves a processed block of decoded time code data (as a struct).
DqAdv650GetTimeRegisters	Retrieves a timekeeper "time register" (sixteen registers).
DqAdv650ProgramPLL	Sets PLL frequency and duty cycle, returns actual frequency set.
DqAdv650ReadEventFifo	Reads data from the event FIFO.
DqAdv650ReadGPS	Reads data from GPS serial FIFO (NMEA format).
DqAdv650SetLocalOffset	Sets the timezone offset, relative to UTC.
DqAdv650SetGPSTime	Sets timekeeper time from GPS module time.
DqAdv650SetPropDelay	Sets the propagation-over-the-wire delay in 10ns increments.
DqAdv650SetTimecodeInput	Configures the input source of the timecode.
DqAdv650SetTimecodeInputEx	Configures the input source of the timecode with custom parameters.
DqAdv650SetTimecodeOutput	Enables or disables timecode output.
DqAdv650SetAMOutputLevels	Sets up AM output levels.
DqAdv650SetAMZCMode	Sets custom parameters for the AM zero-crossing.
DqAdv650ResetTimestampsGetBCD	Latches and returns time captured at the moment of latching it in BCD format and then resets timestmaps on selected layers.
DqAdv650GetTimeANSI	Latches and returns time captured in ANSI/System V format.
DqAdv650SetTimeSBS	Sets time in binary format.
DqAdv650SetTimeANSI	Sets time in ANSI/System V format.
DqAdv650WriteGPS	Writes data into the GPS module serial interface.

3.2 Low-level Programming Techniques

The IRIG-650 or CT-650 offers a simple API to program most of its functions, which is described as follows:

- Time Keeper Programming: shows how to configure the Time Keeper.
- Input Programming: shows how to configure input time codes and use them with the Time Decoder.
- Output programming
- Assigning TTL outputs
- Enabling and disabling subsystems
- Event programming (sync & async): shows how to program the extensive event generation and recording system of the IRIG-650.
- GPS programming
- Calibrating the precision oscillator
- Custom PLL frequency generation

3.2.1 Time Keeper Programming

This section shows how to configure, get & set time for the Time Keeper block described in “Functional Description” on page 6.

3.2.1.1 Configure the Time Keeper

The time keeper can be configured by calling the function:

```
ret = DqAdv650ConfigTimekeeper(hd, devn, mode, flags);
```

For proper functioning, the timekeeper requires a 1PPS signal, which can be delivered from the following sources, specified in <mode> parameter:

1. Internal timebase from the precision oscillator (CT650_TKPPS_INTERNAL)
2. 1PPS clock sources derived from input timecode (CT650_TKPPS_TIME-CODE)
3. 1PPS clock source (CT650_TKPPS_GPS) derived from integrated GPS device (while the GPS device always produces 1PPS pulses, user should read information from GPS device serial port to make sure it has acquired satellites, it is in active state, and its output is synchronized with the satellite clock)
4. Externally from one of the TTL input lines (CT650_TKPPS_IO(N)), where N = [0..3]
5. Externally from RF In1 line (CT650_TKPPS_RFIN)
6. From one of the SYNCx lines (CT650_TKPPS_SYNC(N)), where N = [0..3]
7. Disable timekeeper using CT650_TKPPS_DISABLED mode

There are <flags> worth mentioning:

1. Use internal timebase to generate 1PPS signal required for timekeeping if selected external 1PPS source does not deliver pulses (break in timecode transmission, for example). If the flag CT650_TKFLG_AUTOFOLLOW is selected, timekeeper will switch to internal timebase when an externally derived one is not available. Note that the Time Keeper is programmed to recognize a missing 1PPS if it is delayed for more than 50uS of expected arrival time.
2. Flag CT650_TKFLG_USENOMINAL allows user to select whether to use nominal period (i.e. 100E6 pulses of 100MHz base clock) or the measured

one (Time Keeper measures and averages the number of base clock cycles between externally derived 1PPS pulses when they are valid).

3. If external timebase is not derived from the timecode and if its external timebase is slower than 1PPS, then use the `CT650_TKFLG_SUBPPS` flag.
4. If BCD data in the incoming timecode is corrupted, use the flag `CT650_TKFLG_USESBS` to force the use of SBS for TK hour/min decoding.
5. If you know that seconds, minutes or days information is invalid in the input timecode, use the flags: `CT650_TKFLG_SEC_INVALID`, `CT650_TKFLG_MIN_INVALID` or `CT650_TKFLG_DAY_INVALID`.

This is an example of how to set the Time Keeper for internal timebase:

```
mode = CT650_TKPPS_INTERNAL;
flags = CT650_TKFLG_AUTOFOLLOW|CT650_TKFLG_USENOMINAL;
ret = DqAdv650ConfigTimekeeper(hd, devn, mode, flags);
```

3.2.1.2 Get time from Time Keeper

Once that the Time Keeper is enabled the user can set and read time from it. There are two functions available to read time.

The first function to read time is:

```
ret = DqAdv650GetTimeSBS(hd, devn, &seconds, &micro,
&dayofyear, &year, &tkstatus);
```

This function returns seconds from the beginning of the day, microseconds within the last second at the moment of function call, day of the year, year and the status of the Time Keeper.

The following Time Keeper status conditions are defined:

1. `CT650_TKSTS_BIN2BCD_ERR` - binary time code is invalid (sticky)
2. `CT650_TKSTS_BCD2BIN_ERR` - BCD time code is invalid (sticky)
3. `CT650_TKSTS_TK_ERR` - time code received mismatch current time (sticky)
4. `CT650_TKSTS_1PPS_GEN` - 1pps was generated (sticky)
5. `CT650_TKSTS_1PPS_RCV` - selected 1pps was received (sticky)
6. `CT650_TKSTS_TC_RCV` - external time code was received and applied (sticky)
7. `CT650_TKSTS_1PPS_ADVAL` - 1pps output of adaptive PLL in TK is valid
8. `CT650_TKSTS_PPS_LOST` - 1pps was not detected within last validation interval
9. `CT650_TKSTS_AVG_INVAL` - if averaged 1pps does not pass validation criteria
10. `CT650_TKSTS_1PPS_INVAL` - last detected external 1pps was invalid

Bits marked as “sticky” are set when condition is met and cleared upon read from the status register. Thus, if you call `DqAdv650GetTime…()` in a short period of time between calls, only those bits that were detected will be present on the second call.

`CT650_TKSTS_TK_ERR` is always generated upon synchronization with external timecodes or for IRIG codes with sub-PPS frame rates. For IRIG-B `0x700` code normal, for sub-PPS and high frame rate codes, 1PPS invalid warnings might be set.

The second function call to read time:

```
ret = DqAdv650GetTimeANSI(hd, devn, &time, &micro, &tkstatus);
```

returns locally correct time from the timekeeper in a standard Unix format (defined in “time.h”):

```
typedef struct {
   int tm_sec;     /* seconds after the minute - [0,59] */
   int tm_min;     /* minutes after the hour - [0,59] */
   int tm_hour;    /* hours since midnight - [0,23] */
   int tm_mday;    /* day of the month - [1,31] */
   int tm_mon;     /* months since January - [0,11] */
   int tm_year;    /* years since 1900 */
   int tm_wday;    /* days since Sunday - [0,6] */
   int tm_yday;    /* days since January 1 - [0,365] */
   int tm_isdst;   /* daylight savings time flag */
} CT_ANSI_TIME, *pCT_ANSI_TIME
```

3.2.1.3 Set the Local Time Offset

To set local time offset from UTC time, call:

```
ret = DqAdv650SetLocalOffset(hd, devn, hour_offset, min_offset);
```

and provide signed offset in hours and minutes within the hour. Once local time is set, firmware will convert local time into UTC time to store in Time Keeper’s registers and account for it when time is read. The local offset must be set before setting or requesting the Time Keeper time.

The two functions: `DqAdv650GetTimeBCD` and `DqAdv650GetTimeRegisters` read time straight from time decoder registers and:

- Do not provide local time correction
- Do not work if time decoder continuously decodes a valid input timecode

3.2.1.4 Set time for Time Keeper

Either `DqAdv650SetTimeANSI` or `DqAdv650SetTimeSBS` can be used to set the Time Keeper time.

The first is easier to use (this example includes call to POSIX time functions):

```
time_t timer;
time_t seconds;
struct tm * tmi;

// request local time from computer
seconds = time(&timer);
tmi = localtime(&timer);

// apply this time to the timekeeper
ret = DqAdv650SetTimeANSI(hd, devn, 0, tmi, &status);
```

The second function call allows you to set time as binary seconds:

```
ret = DqAdv650SetTimeSBS(hd, devn, flags, seconds,
                         year_day, year, &status);
```

Select <`flags`> from the following:

`CT650_TIME_TM_SEC` – seconds of the day are valid
`CT650_TIME_TM_YEAR` – year is valid
`CT650_TIME_TM_YDAY` – day of the year is valid

to specify what part of the supplied time is valid.

3.2.2 Input Programming

When programming the Time Decoder, the first thing you'll need to do is to select a timecode as a time source of the Time Keeper (TK). Otherwise, it will not receive a timecode from the input stream.

We've seen in the previous section that we program the Time Keeper by calling the function `DqAdv650ConfigTimekeeper()`; for example:

```
mode = CT650_TKPPS_TIMECODE; // TK mode: ext, <= 1PPS
flags = CT650_TKFLG_AUTOFOLLOW|CT650_TKFLG_USENOMINAL;
ret = DqAdv650ConfigTimekeeper(hd, devn, mode, flags);
```

Then, the Time Decoder needs to be programmed:

```
mode = CT650_IN_ENABLED;     // timecode input: enable
ret = DqAdv650SetTimecodeInput(hd, devn, mode, td_input,
pPrmDef, pDataDef);
```

The input <td_input> can be taken from one of the following sources:

- `CT650_IN_AM` - Time code is an amplitude modulated (AM) signal
- `CT650_IN_M2_RF0` - Manchester II code on RFIn0 (Ext Clk In) input
- `CT650_IN_M2_RF1` - Manchester II code on RFIn1 (Rxt Trig In) input
- `CT650_IN_M2_IO0` - Manchester II code on TTLIn0 input
- `CT650_IN_M2_IO1` - Manchester II code on TTLIn1 input
- `CT650_IN_NRZ_RF0` - NRZ code on RF0 (Ext Clk In) input
- `CT650_IN_NRZ_RF1` - NRZ code on RF1 (Ext Clk In) input
- `CT650_IN_NRZ_IO0` - NRZ code on TTLIn0 input
- `CT650_IN_NRZ_IO1` - NRZ code on TTLIn1 input

The parameters <pPrmDef> and <pDataDef> are explained in the next section.

3.2.2.1 Timecode definitions (for input)

The following two parameters define the type of data expected in the timecode. The first one is `pPrmDef`, which defines timecode parameters (see DaqLibHL-TimeCodes.h):

```
// Timecode Parameters
typedef struct {
    char tcode_name[32];    // timecode name
    char tcode_subtype[32]; // timecode signal
    double bit_frq;         // timecode bit rate, Hz
    double frame_frq;       // timecode frame rate, Hz
    double carrier_frq;     // carrier frequency
    uint32 ratio_f_er;      // frequency to code bit ratio
    uint32 logic_0;         // number of cycles for logic_0
    uint32 logic_1;         // number of cycles for logic_1
    uint32 pos_id;   // num cycles for position identifier
    int is_bcd;      // TRUE if BCD code is present in data
    int is_cf;       // TRUE if CF codes are present
    int is_sbs;      // TRUE if SBS is present
   int is_year; //TRUE if CF chars 50-58 have BCD year data
} CT650_IRIG_PRM_DEF, *pCT650_IRIG_PRM_DEF;
```

For example, the time code IRIG-B B004/B124 is defined as follows:

```
// IRIG B -- IRIG parameters structure definition
CT650_IRIG_PRM_DEF irig_b_bcd_cf_sbs_prm = {
    "IRIG B",    // encoding name
    "B124",      // signal subtype
    100.0,       // 100Hz bit rate
    1.0,         // 1Hz frame rate
    1000.0,      // 1kHz carrier frequency
    10,          // 10 cycles per bit
    2,           // two "high" for logic 0
    5,           // five "high" for logic 1
    8,           // eight "high" for position identifier
    TRUE,        // TRUE if BCD code is present in the data
    TRUE,        // TRUE if CF codes are present
    TRUE,        // TRUE if SBS is present
    TRUE         // TRUE if CD 50-58 contain BCD year data
};
```

The **encoding name** and **signal subtype** are used for identification purposes only and are not cross-checked internally against content of the data table and/or other paramters.

Bit rate characterizes number of characters to be received in one second.

Frame rate specifies how often the code is repeating itself – in case of most popular IRIG-B code it's once per second.

Carrier frequency applies to both DCLS (direct current level shift, also known as NRZ-L code) and AM timecodes. In case of AM, it is the frequency of the sine signal. In case of DCLS, it is quantization frequency.

Number of **cycles per bit** is the character length in carrier periods; here it is ten. For character "0", each character consists of two "high" and eight "low" periods. For character "1", each character consists of five "high" and five "low" periods. For **position identifier**, each char consists of eight "high" and two "low" periods. A character consisting of ten periods of the same level (normally "low") is considered idle character and is rarely used.

The last four fields of the structure overrides information in the data definition table and depends on whether actual time code contains **BCD time** information, control functions fields (**CF**), straight binary seconds (**SBS**) or **year** information in CF field character. The data definition tables are prepackaged by assumption that all this information is available in the input or output timecode and these bits can quickly disable or enable named timecode blocks in accordance with the actual data.

UEI provides pre-packaged code definitions for IRIG-A/B/D/E/G/H which can be tweaked by the user to match their actual time code as in this example:

```
pCT650_IRIG_PRM_DEF pOutPrm;
pOutPrm =
(pCT650_IRIG_PRM_DEF)malloc(sizeof(CT650_IRIG_PRM_DEF));
memcpy(pOutPrm, pPrmDef, sizeof(CT650_IRIG_PRM_DEF));

pOutPrm->is_sbs = FALSE; // supress SBS in output
ret = DqAdv650SetTimecodeOutput(hd, devn, mode, output,
pOutPrm, pDataDef);
```

`pDataDef` defines timecode data and the internal location of this data in the registers:

```
typedef struct {
    char timecode_name[32]; // timecode name
    uint32 val_table_sz;    // timecode table size
    // timecode table (128 bits maximum)
    CT650_VALID_TABLE val_table[CT650_VAL_MAX];
} CT650_IRIG_CODE_DEF, *pCT650_IRIG_CODE_DEF;
```

The maximum size of the table is 128 characters while the longest code used is 120 characters.

The timecode is represented by `CT650_VALID_TABLE` structure:

```
typedef struct {
    uint16 cmd;      // command
    uint16 bit;      // bit number (in TK TREGx)
    uint16 reg;      // TREG number
    uint16 desc;     // data description
} CT650_VALID_TABLE, *pCT650_VALID_TABLE;
```

For example, IRIG-B code is described as follows:

```
CT650_IRIG_CODE_DEF irig_b_bcd_cf_sbs = {
  "IRIG B",
  100,
  {
    //  Command,      Bit,       Register,    Description

    // == BCD seconds: ==
    // position 99 P0
    { CT650_VAL_P, BCD_NoP_b(), BCD_NoP_r(), BCD_NoP_d()},
    // position 0  Pr
    { CT650_VAL_P, BCD_NoP_b(), BCD_NoP_r(), BCD_NoP_d()},
    // position 1  unit seconds
    { CT650_VAL_01,BCD_SEC_b(0),BCD_SEC_r(0),BCD_SEC_d(0)},
    // position 2
    { CT650_VAL_01,BCD_SEC_b(1),BCD_SEC_r(1),BCD_SEC_d(1)},
    // position 3
    { CT650_VAL_01,BCD_SEC_b(2),BCD_SEC_r(2),BCD_SEC_d(2)},
    // position 4
    { CT650_VAL_01,BCD_SEC_b(3),BCD_SEC_r(3),BCD_SEC_d(3)},
    // position 5
    { CT650_VAL_NOP,BCD_NoP_b(),BCD_NoP_r(), BCD_NoP_d() },
    // position 6 tens seconds
    { CT650_VAL_01,BCD_SEC_b(4),BCD_SEC_r(4),BCD_SEC_d(4)},
    // position 7
    { CT650_VAL_01,BCD_SEC_b(5),BCD_SEC_r(5),BCD_SEC_d(5)},
    // position 8
    { CT650_VAL_01,BCD_SEC_b(6),BCD_SEC_r(6),BCD_SEC_d(6)},

    // == BCD minutes: ==
    // position 9  P1
    { CT650_VAL_P, BCD_NoP_b(), BCD_NoP_r(), BCD_NoP_d()},
    // position 10 unit minutes
    { CT650_VAL_01,BCD_MIN_b(0),BCD_MIN_r(0),BCD_MIN_d(0)},
…
```

The time code always starts with two **position** identifiers (rarely used legacy timecodes use a single position identifier which can be accommodated by setting input `mode` with the `CT650_IN_P0_ONLY` flag). The Time Decoder waits until two position identifiers come one after another and then begin decoding symbols one by one.

The **command** field identifies which type of character is expected to be received in this position. Following characters are valid for input:

```
CT650_VAL_NOP       0  // 000 - NOP, go to the next
                       // character in the message
CT650_VAL_01        4  // 100 - "0" or "1" is expected,
                       // copy to target register
CT650_VAL_P         5  // 101 - Px marker is expected
CT650_VAL_0         6  // 110 - "0" is expected
CT650_VAL_1         7  // 111 - "1" is expected
```

The **Bit** field can be one of the following:

```
#define BCD_NoP_b()       (0)   //No operation for BCD section
#define BCD_SEC100_b(B)   ((B))   // 1/100 of a second
#define BCD_SEC10_b(B)    ((B)+8) // 1/10 of a second
#define BCD_SEC_b(B)      ((B))   // seconds
#define BCD_MIN_b(B)      ((B)+8) // minutes
#define BCD_HOUR_b(B)     ((B))   // hours
#define BCD_DAYS_b(B)     ((B))   // days
#define BCD_YEAR_b(B)     ((B))   // year
#define BCD_LEAPYR_b()    ((8))  // leap year flag
#define BCD_DLFLAG_b()    ((9)) // DL flag
// SBS_SEC_b(B): 17th SBS second bit is stored in TREG2/BIT15
#define SBS_SEC_b(B)      (((B)<=15)?(B):(15))
#define CF_NoP_b()        (0)  // No operation for CF section
#define SBS_NoP_b()       (0)  // No operation for SBS section
```

The **Bit** field defines the bit position in the time registers (TREGs) table where this data is stored.

The **Register** field defines in which register the data is located:

```
#define BCD_NoP_r()       (0)  // No operation for BCD section
#define BCD_SEC100_r(B)   ((0))// 1/100 of a second
#define BCD_SEC10_r(B)    ((0))// 1/10 of a second
#define BCD_SEC_r(B)      ((1))// seconds
#define BCD_MIN_r(B)      ((1))// minutes
#define BCD_HOUR_r(B)     ((2))// hours
#define BCD_DAYS_r(B)     ((3))// days
#define BCD_YEAR_r(B)     ((5))// year
#define BCD_LEAPYR_r()    ((2))// leap year flag
#define BCD_DLFLAG_r()    ((2))// No operation for CF section
#define SBS_SEC_r(B)           // SBS seconds register
#define CF_NoP_r()        (0)  // No operation for CF section
#define SBS_NoP_r()       (0)  // No operation for SBS section
```

The **Description** field defines the type of data stored in this character:

```
                                // Sections of data <sec>
#define CT650_SECT_BCD  1       // BCD time data
#define CT650_SECT_CF   2       // control functions
#define CT650_SECT_SBS  3       // straight binary seconds

#define BCD_NoP_d()         (CT650_SECT_BCD<<8) //NOP or Positional
character

#define BCD_SEC100          1
#define BCD_SEC100_d(B)     ((BCD_SEC100<<12)|(CT650_SECT_BCD<<8)|(B))

#define BCD_SEC10           2
#define BCD_SEC10_d(B)      ((BCD_SEC10<<12)|(CT650_SECT_BCD<<8)|(B))

#define BCD_SEC             3
#define BCD_SEC_d(B)        ((BCD_SEC<<12)|(CT650_SECT_BCD<<8)|(B))

#define BCD_MIN             4
#define BCD_MIN_d(B)        ((BCD_MIN<<12)|(CT650_SECT_BCD<<8)|(B))

#define BCD_HOUR            5
#define BCD_HOUR_d(B)       ((BCD_HOUR<<12)|(CT650_SECT_BCD<<8)|(B))

#define BCD_DAYS            6
#define BCD_DAYS_d(B)       ((BCD_DAYS<<12)|(CT650_SECT_BCD<<8)|(B))

#define BCD_YEAR            7
#define BCD_YEAR_d(B)       ((BCD_YEAR<<12)|(CT650_SECT_BCD<<8)|(B))

#define BCD_SPEC_BIT        0
#define BCD_LEAPYR_d()      ((BCD_SPEC_BIT<<12)|(CT650_SECT_BCD<<8)|0)

#define BCD_DLFLAG_d()      ((BCD_SPEC_BIT<<12)|(CT650_SECT_BCD<<8)|1)
#define SBS_SEC_d(B)        ((BCD_SPEC_BIT<<12)|(CT650_SECT_SBS<<8)|(B))
#define CF_BITS_d(B)        ((BCD_SPEC_BIT<<12)|(CT650_SECT_CF<<8)|(B))
#define CF_NoP_d()          (CT650_SECT_CF<<8)  //NOP or Positional char
#define SBS_NoP_d()         (CT650_SECT_SBS<<8) //NOP or Positional char
```

TREGs are logic registers where time data is collected. There are two sets of registers and when `ret = DqAdv650GetInputTimecode(hd, devn, tcsize, &block, decoded_code);` is called, it returns data from the inactive set of registers (i.e. the ones that are not currently processing input timecode).

3.2.2.2 Input from the Time Decoder

Two functions are available to receive information directly from the time decoder. Mentioned earlier, `DqAdv650GetInputTimecode()` returns a string of received characters along with timestamps when these characters were received. This function is very important for debugging the input timecode:

```
// Read raw time code from timecode memory
ret = DqAdv650GetInputTimecode(hd, devn, tcsize, &block,
decoded_code);
for (i = 0; i < tcsize; i++) {
    uint32 tcode = 0x3 & decoded_code[i];
    if (0 == (i%10)) printf(".");
    switch (tcode) {
        case 0: printf("i"); break;
        case 1: printf("P"); break;
        case 2: printf("0"); break;
        case 3: printf("1"); break; }
}
```

<tcsize> should be set to accommodate all characters in the timecode (for example IRIG-B has 100 characters in the code).

Decoded data is represented as follows:

Bit	Name	Description	Reset state
31-2	SP_RSV	For first two characters – "Special bits": Character 0 : layer's timestamp bits [29:0] (in 10us intervals) Character 1 : phase delay of the input (bits [29:0]) For all other characters - reserved	0
1-0	CHAR	Decoded characters CT650_CHAR_IDLE=0 Idle character (0% duty cycle) CT650_CHAR_POS=1 Position Identifier (80% duty cycle typical) CT650_CHAR_ZERO=2 Logic "0" (20% duty cycle typical) CT650_CHAR_ONE=3 Logic "1" (80% duty cycle typical)	0

The second function DqAdv650GetTimeRegisters() returns content of sixteen time registers as an array of sixteen uint32's:

```
ret = DqAdv650GetTimeRegisters(hd, devn, time_regs);
printf("--TREGx S10:%x M/S:%x H:%x D:%x SBS:%x Y:%x\n",
time_regs[0], time_regs[1], time_regs[2], time_regs[3],
time_regs[4], time_regs[5]);
```

The structure of the time registers (TREG) is:

Address	Register	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x2100	TREG0					BCD 1/10s of the seconds								BCD 1/100s of the seconds			
0x2104	TREG1		BCD Minutes								BCD Seconds						
0x2108	TREG2	SBS[16]						DL flag	Leap year		BCD Hours						
0x210C	TREG3							BCD days									
0x2110	TREG4	Straight Binary Seconnds (SBS) [15:0]															
0x2114	TREG5																
0x2118	TREG6																
0x211C	TREG7																
0x2120	TREG8																
0x2124	TREG9																
0x2128	TREG10																
0x212C	TREG11																
0x2130	TREG12	Phase delay MSB (value of the CT650_DBCNT)															
0x2134	TREG13	Phase delay LSB (value of the CT650_DBCNT)															
0x2138	TREG14	Timestamp MSB (value of the layer's timestamp register)															
0x213C	TREG15	Timestamp LSB (value of the layer's timestamp register)															

The Firmware takes care of setting proper parameters for the analog input part of the Time Decoder based on the timecode parameters selected. One other function available to advanced users is:

```
int DqAdv650SetAMZCMode(int hd, int devn, uint32 flags, int
zc_adjust)
```

This function can override calculated setting made by the firmware. It needs to be called after DqAdv650SetTimecodeInput() with the CT650_IN_DISABLED flag, but before enabling layer input with DqAdv650Enable().

Calling `DqAdv650SetAMZCMode` configures AM zero-crossing parameters:

- `CT650_ZCCFG_ZAV(N)` – number of samples in average for ADC data
- `CT650_ZCCFG_ZCAZ` – =1-use "auto zero" if it auto-calculated zero level is within of the 1/8 of "zero_level" (ZCL_Dx)
- `CT650_ZCCFG_ZCZD` – Select direction of the crossing for min/max detection (1=rising)
- `CT650_ZCCFG_ZCAS(N)` – 0=1, 1=2, 2=4, 3=8, 4=16, 5=32, 6=64, 7=128, 8=256
- `CT650_ZCCFG_ZCSC(N)` – 0=1, 1=2, 2=4, 3=8, 4=16, 5=32, 6=64, 7=128, 8=256

To completely take control over input timecode parameters, use this call:

```
int DqAdv650SetTimecodeInputEx(int hd, int devn, uint32 mode,
                               uint32 input,
                               pCT650_IRIG_PRM_DEF pPrmDef,
                               pCT650_IRIG_CODE_DEF pDataDef,
                               pCT650_IRIG_SIG_DEF pSigDef,
                               uint32 sigDefMask);
```

<pPrmDef> and <pDataDef> need to be set to the same tables as in `DqAdv650SetTimecodeInput()`, however most of the automatically calculated parameters can be overwritten. Additionally, <pSigDef> is defined as:

```
typedef struct {
    // user-defined part
    double ppx;         // time between time codes  (for
                        // most IRIG-x == 1pps)
    uint32 pph;         // pulses per hour, 1PPS=3600PPH
    uint32 in_freq;     // input carier frequency
    uint32 rx_ici;      // ct650_rx_ici - pulses per bit
    uint32 dbcnt;       // ct650_dbcnt (debouncing counter
                        // only in NRZ input mode in 10ns)
    uint32 rxmsg_len;   // ct650_rxmsg_len input message
                        // length
    uint32 pi_min;      // ct650_pi_min - minimum number of
                        // pulses in position identifier
    uint32 pi_max;      // ct650_pi_max - maximum number of
                        // pulses in position identifier
    uint32 l0_min;      // ct650_l0_min - minimum number of
                        // pulses in level 0 position
    uint32 l0_max;      // ct650_l0_max - maximum number of
                        // pulses in level 0 position
    uint32 l1_min;      // ct650_l1_min - minimum number of
                        // pulses in level 1 position
    uint32 l1_max;      // ct650_l1_max - maximum number of
                        // pulses in level 1 position

    // system-defined part
    uint32 cr_tol_min;  // ct650_cr_tol_min (signal
                        // tolerance - MAN2 mode only)
```

```
    uint32 ct_tol_max;  // ct650_ct_tol_max (signal
                        // tolerance - MAN2 mode only)
    uint32 mii_tol;     // ct650_mii_tol 50us @ 100Mhz base
    uint32 pps_min;     // ct650_pps_min - flywheel clock
                        //divider auto-correction low limit
    uint32 pps_max;     // ct650_pps_max - flywheel clock
                        //divider auto-correct'n high limit
    uint32 ppsmav;      // ct650_ppsmav 3 = 8 points
} CT650_IRIG_SIG_DEF, *pCT650_IRIG_SIG_DEF
```

To tell the function which of those parameters needs to be overwritten, use one of more of the following <flags>:

```
// constant flags to validate each
// member of CT650_IRIG_SIG_DEF
#define CT650_RX_ICIWR      (1L<<0)
#define CT650_DBCNTWR       (1L<<1)
#define CT650_RXMSG_LENWR   (1L<<2)
#define CT650_PI_MINWR      (1L<<3)
#define CT650_PI_MAXWR      (1L<<4)
#define CT650_L0_MINWR      (1L<<5)
#define CT650_L0_MAXWR      (1L<<6)
#define CT650_L1_MINWR      (1L<<7)
#define CT650_L1_MAXWR      (1L<<8)
#define CT650_CR_TOL_MINWR  (1L<<9)
#define CT650_CR_TOL_MAXWR  (1L<<10)
#define CT650_MII_TOLWR     (1L<<11)
#define CT650_PPS_MINWR     (1L<<12)
#define CT650_PPS_MAXWR     (1L<<13)
#define CT650_PPSMAVWR      (1L<<14)
```

3.2.2.3 Fine-tuning Input Parameters & Registers

The following function may be useful for fine-tuning input timecode parameters outside of the common timecodes prepackaged for ease of use. Since the use of this function requires deep knowledge of processes inside time decoding algorithms and trial-and-error approach, the IRIG-650 API provides a special low-level function to read internal registers:

```
int DAQLIB DqAdv650ReadReg(int hd, int devn, uint32 reg,
uint32* value);
```

Information from the following registers is most useful to find problems with the input AM timecode signal (ADC produces 2s complement 16-bit data):

```
#define CT650_ADC_AV     0x2010    // Average of ADC data
// Calculated level of zero (16-bit)
#define CT650_ZCL        0x2084
// Averaged "High"("Mark") amplitude minimum
#define CT650_ZC_AMMAX   0x2088
// Averaged "High"("Mark") amplitude maximum
#define CT650_ZC_AMMIN   0x208C
// Zero crossing "High"("Mark") amplitude maximum
#define CT650_ZC_MMAX    0x2090
// Zero crossing "High"("Mark") amplitude minimum
#define CT650_ZC_MMIN    0x2094
// Zero crossing "Low"("Space") amplitude maximum
#define CT650_ZC_SMAX    0x2098
// Zero crossing "Low"("Space") amplitude minimum
#define CT650_ZC_SMIN    0x209C
```

Contact our technical support for other types of low-level information available for debugging purposes.

Since there is a propagation delay for the signal to travel across a wire, you can use `DqAdv650SetPropDelay()` to adjust propagation delay of the timecode signal in the wiring with 10ns resolution (applies onto to DCLS or MII signaling).

3.2.3 Output programming

For output data definition, the data definition table is also represented in terms of TREG addressing, and the firmware automatically translates it into an output format table.

Output automatically appears on AM output line and DCLS signal needs to be routed to TTL Out lines (see“Layer Connectors and Wiring” on page 9).

To configure the output timecode, use `DqAdv650SetTimecodeOutput()`. As an example, we use the same timecode parameter and data definition table as in the input example, but disable the straight binary seconds (SBS) section:

```
pOutPrm->is_sbs = FALSE;
ret = DqAdv650SetTimecodeOutput(hd, devn, mode, output,
pOutPrm, pDataDef);
```

`<mode>` can be a combination of the following bits:

- `CT650_OUT_DISABLED` – start output in disabled state
- `CT650_OUT_ENABLED` – start output in enabled state
- `CT650_OUT_ONVALID` - Disable output waveform generation until time assembler restarts (may take two minutes for sub-PPS codes like IRIG-E or H) to avoid incorrect output during initial synchronization with external time signal source.
- `CT650_OUT_EXTSYNCRQ` - External Resynchronization required bit – works as a modifier for the `CT650_OUT_ONVALID`, if this bit is set and `CT650_OUT_ONVALID` as well output waveform is disabled until resynchronization with external source is achieved plus one ta_restart pulse is received from the time keeper (extra ta_restart period is required because time keeper re-syncs in the middle of the pps/ppm/pph period. Note that for time codes that repeat every hour (IRIG-D) the re-synchronization process can take up to three hours.

For example, to start output when input timecode is valid:

```
mode = CT650_OUT_ONVALID |   // start output when input
       CT650_OUT_EXTSYNCRQ;  //  timecode is valid
```

Calling `DqAdv650SetAMOutputLevels(int hd, int devn, uint32 flags, double hi_gain, double low_gain, double offset)` allows you to adjust the AM output level, low and high signal ratio and signal DC offset.

`<hi_gain>` adjusts "logic 1" gain [0..2]

`<low_gain>` adjusts "logic 0" gain [0..2]

`<offset>` adjusts DC output offset [-0.5..+0.5]

The default high-to-low ratio is 3:1 with zero offset and 2V amplitude.

3.2.4 Assigning TTL outputs

The IRIG-650 board has four TTL-level output lines which can be assigned to variety of digital signals available inside the card.

```
mode = CT650_OUT_TTLEN1;
ret = (hd, devn,
                                mode,    // mode of operation
                                0,       // output configuration
                                CT650_OUT_CFG_SRC_1PPS,  // TTL0
                                CT650_OUT_CFG_SRC_NRZ,   // TTL1
                                CT650_OUT_CFG_SRC_1PPM,  // TTL2
                                CT650_OUT_CFG_SRC_1PPH); // TTL3
```

There are certain important flags to use for the mode of operation:

To shorten pulse duration to 40ns (the default is 60µs, shorter pulses need to be used for any signal with output frequency in excess of 5kHz), set by these flags:

`CT650_OUT_TTL3_40nS`

`CT650_OUT_TTL2_40nS`

`CT650_OUT_TTL1_40nS`

`CT650_OUT_TTL0_40nS`

`CT650_OUT_TTL_40nS`

50 Ohm termination is available on three input lines, set by these flags:

`CT650_OUT_RF1TERM` - Enable 50Ω termination for TTL RF1 (TRIGIN) input

`CT650_OUT_RF0TERM` - Enable 50Ω termination for TTL RF0 (CLKIN) input

`CT650_OUT_AMTERM` - Enable 50Ω termination for the AM input

One or two buffers should be enabled to drive TTL output lines (TTL0 to TTL3):

`CT650_OUT_TTLEN1` - Enable buffer 1 (applies to all lines)

`CT650_OUT_TTLEN0` - Enable buffer 0 (applies to all lines)

Each of the four programmable TTL outputs can be programmed to be driven by one of the following 32 available sources:

`CT650_OUT_CFG_AM_NRZ` - IRIG-AM-IN -> NRZ-L/DCLS output (debug)

`CT650_OUT_CFG_GPS_FIXV` - GPS Fix_Valid output

`CT650_OUT_CFG_GPS_ANSH` - GPS Antenna Shorted output

`CT650_OUT_CFG_GPS_ANOK` - GPS Antenna Ok output

`CT650_OUT_CFG_GPS_TXD1` - GPS TXD1 (COM1) output

`CT650_OUT_CFG_GPS_TXD0` - GPS TXD0 (COM0) output

`CT650_OUT_CFG_SRC_1US` - 0.000001sec pulse

`CT650_OUT_CFG_MII_NRZ` - Manchester II -> NRZ sequence (reserved)

`CT650_OUT_CFG_EVENT3` - Event 3 (on event: 60µs or 1ms pulse)

`CT650_OUT_CFG_EVENT2` - Event 2

`CT650_OUT_CFG_EVENT1` - Event 1

`CT650_OUT_CFG_EVENT0` - Event 0

CT650_OUT_CFG_SRC_SYNC3 - Drive output from sync[3]
CT650_OUT_CFG_SRC_SYNC2 - Drive output from sync[2]
CT650_OUT_CFG_SRC_SYNC1 - Drive output from sync[1]
CT650_OUT_CFG_SRC_SYNC0 - Drive output from sync[0]
CT650_OUT_CFG_CR_OUT - Output carrier frequency
CT650_OUT_CFG_SRC_CR - Custom frequency output (from PLL)
CT650_OUT_CFG_SRC_10MHZ - Precision 10MHz
CT650_OUT_CFG_SRC_5MHZ - Precision 5MHz
CT650_OUT_CFG_SRC_1MHZ - Precision 1MHz
CT650_OUT_CFG_SRC_NRZS - Output DCLS / NRZ start strobe
CT650_OUT_CFG_SRC_MII - Manchester-II output time code
CT650_OUT_CFG_SRC_NRZ - DCLS/NRZ-L output time code
CT650_OUT_CFG_SRC_1GPS - Re-route GPS 1PPS pulse
CT650_OUT_CFG_SRC_1PPH - 1PPH pulse
CT650_OUT_CFG_SRC_1PPM - 1PPM pulse
CT650_OUT_CFG_SRC_1PPS - 1PPS pulse
CT650_OUT_CFG_SRC_0_1S - 0.1sec pulse
CT650_OUT_CFG_SRC_0_01S - 0.01sec pulse
CT650_OUT_CFG_SRC_1 - Drive output with 1
CT650_OUT_CFG_SRC_0 - Drive output with 0

Note that to output a time code (NRZ/MII) both the Time Encoder & Decoder must have already been configured as in the above section.

DqAdv650AssignTTLOutputs can also be used to re-route signals to the SYNCx line (internal synchronization lines between the layers in the same IOM):

```
ret = DqAdv650AssignTTLOutputs(hd, devn,
CT650_OUT_SYNCEN0|CT650_OUT_SYNCEN1|CT650_OUT_SYNCEN2|
CT650_OUT_SYNCEN3,
DQ_CT650_TTL_SYNCX,
CT650_OUT_CFG_SRC_1PPS, CT650_OUT_CFG_SRC_1PPS,
CT650_OUT_CFG_SRC_1PPS, CT650_OUT_CFG_SRC_1PPS);
```

CT650_OUT_SYNCENx constants are used to enable a SYNC line for output:

```
#define CT650_OUT_SYNCEN3   (1L<<27)        // =1 -
Enable SYNC3 buffer
#define CT650_OUT_SYNCEN2   (1L<<26)        // =1 -
Enable SYNC2 buffer
#define CT650_OUT_SYNCEN1   (1L<<25)        // =1 -
Enable SYNC1 buffer
#define CT650_OUT_SYNCEN0   (1L<<24)        // =1 -
Enable SYNC0 buffer
```

The current state of SYNC line can be routed to any TTL Out line using one of the CT650_OUT_CFG_SRC_SYNCx constants.

NOTE: Make sure that no two layers actively are driving the same SYNC line simultaneously.

3.2.5 Enabling and disabling subsystems

Both input and output can be initialized in the enabled or disabled state. Since all subsystems eventually synchronize to the same time it doesn't make a significant difference when subsystems are enabled.

To configure a subsystem in enabled state, so that it runs immediately after the function call, set:

```
mode |= CT650_OUT_ENABLED;
```

in `DqAdv650SetTimecodeInput()` or `DqAdv650SetTimecodeOutput()`.

If subsystem needs to be configured but prevented from running, use:

```
mode |= CT650_OUT_DISABLED;
```

In the latter case, user should call

```
ret = DqAdv650Enable(hd, DEVN, TRUE);
```

to enable initialized subsystems together, and

```
ret = DqAdv650Enable(hd, DEVN, FALSE);
```

to disable them.

3.2.6 Event programming (sync & async)

The IRIG-650 implements an extensive event generation and recording system.

There are two ways of using events:

1. Synchronous events: when the user configures events and then either reads event status registers, or reads events stored in the event FIFO. Additionally, events can be routed to TTL Out or SYNCx lines to be used for synchronization and time-related purposes. Synchronous events can be used to trigger selected layers to start or stop acquisition at an exact time provided by an IRIG-B source (or time extracted from a GPS signal).
2. Asynchronous events: when the user configures events to be received asynchronously and then waits for the event packet (on a dedicated UDP port that is different from the main UDP port where most other traffic goes). Asynchronous events are mainly used to drive real-time systems from precise clocks as well as to inform the user about unexpected timecode errors. On a Windows XP SP3 platform running on a 2GHz Intel Core2 Quad CPU, a measured delay of 80µs between time event and thread invocation was achieved 99.5% of the time. The actual performance may vary depending on the performance of the host system, system load, and the type of IOM used (MPC5200 – 100-base-T or MPC8347 – 1000-Base-T)

The IRIG-650 layer can use four Event modules (as channels 0 to 3) to create timed interrupts and provide synchronization signals to the outside world. Additionally, a dedicated event module is used to provide clocking to the input channel list, allowing it to save timestamp information with 10µs resolution by default (`DQ_LN_10us_TIMESTAMP` with the `DQL_TMRCFG_TSTS_66M` source) or to a custom resolution by using the `DqCmdResetTimestamp` function.

Each event can generate up to two sub-events, each with programmable delay, the main event starts a countdown (delay) after which each sub-event is fired. Events may be set in one-time (alarm) or repeat mode.

For special logics, by using masks, events can repeat every year/day/hour/minute or second; they can also be triggered by a special condition on the DNA bus and/or global start/stop trigger.

Table 3-1 provides a summary of event registers. Registers and their bits are represented by color groups that differ for different event sources.

3.2.6.1 Programming Events

To program events, pass the `<event>` structure as an argument to the function: `DqAdv650SetEvents(hd,devn, evt_chan, flags, &event, &param)` for which the valid event channel `<evt_chan>` can be set to 0,1,2, or 3. The `<event>` structure is defined as:

```
typedef struct {
    uint32 event_cfg;       // event configuration bitfield
    uint32 event_prm;       // event parameters
    uint32 event_val;       // parameter-dependent value
    uint32 event_sub0_dly;  // subevent 0 delay
    uint32 event_sub1_dly;  // subevent 1 delay
} EV650_CFG, *pEV650_CFG;
```

Table 3-1 summarizes possible structure values that depend on the mode of operation. The structure values are described on the following pages. Notice that the same fields can have a different meaning based on the `<event_cfg>` mode of operation.

Table 3-1. Event Registers Summary

<event_cfg>

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EN	DLYC	STPE	EV1IRQ	EV0IRQ	DBL	RPT	EVTPL	Event pulse length in clocks				ISRC		STTE	IRSRC					YM	DM	HM	MM	SM	UM	EDGE	ESRC				
1- enable event, auto-cleared for single events	Delay clock 0=1MHz 1=66/100M	1-use stop trigger when disarm	Issue event 1 detected	Issue event DNA IRQ when sub-event 0 detected	Dual 0-single 1-dual	Repeat 1-repeat	0 1-13 14 15	Valid for dual events only, PWM mode, event output =1 after subevent0 0-one-time and goes back to 0 after subevent1 2-14 66/100MHz clocks				Internal counter source selector ISRC#=0: 00 - 66/100MHz 10 - 1 MHz 11 - 1KHz 00 - PLL 10 - varies, on CT-650 - 1MHz from TK, synchronized with 1 PPS		1-use start trigger to arm event	Internal counter reset source - same codes as for the event source, can be used for watchdog implementation or event re-synchronized by external source, 1PPS, etc Used when ESRC=4 (will reset internal counter) and ESRC=6 (will set time interval for DPLL)					Year mask (SB time only)	Day mask mask	Hour mask	Minutes mask SRC# ISRC field clock modifier 0-Internal 1-External	Seconds mask	Micro-seconds mask	Edge 0-falling 1-rising	Event source 0 Disabled 1 Software only 2 BCD time 3 SB time 4 Internal counter 5 DNA bus 6 DPLL - digital PLL 7 Reserved 8-31 Digital events				

All events are reported as a test IRQ in the main DNA IRQ register (IR_TI, 1<<21)

<event_prm> WR: various mode-dependent configuration parameters/RD: timestamp capture for the digital events

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ICNTDIV	Internal counter divider - for the ESRC=4																														
								DPLLF	DPLL frequency (# of pulses between two intervals set by IRSRC source) - for the ESRC=6																						
				SEC	BCD seconds (ESRC=2)							MIN	BCD minutes (ESRC=2)			HR	BCD hour (ESRC=2)					DOY	BCD day of year (ESRC=2)								
			SBS	Straight binary seconds of the day for the SB time mode (ESRC=3)																SBY	Straight binary year for the SB time mode (ESRC=3)											
LSA	Layer address (DNA only)			DRD	DWR	DCS		DDLY # of DNA clocks after read				DNAAV # of valid DNA address MSBs				DNAA DNA address														DLAC	DDC
				Monitor	Monitor	DNA CS#		or write prior to compare data				0 - ignore address (monitor data only)																		1-use layer address	1-compare DNA data
				DNA reads	DNA writes	to monitor						1-14 - select# of valid bits																			
TS	Timestamp capture register for the digital events only - stores timestamp that is corresponding to the digital event detection time																														

<event_val> various mode-dependent configuration parameters

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SBD	Straight binary day of the year for the SB time mode (ESRC=3)											USEC	Micro seconds (ESRC=2 or 3)																		
DNADC	DNA data "COMPARE" value - for the ESRC=11																														

<event_delay0> Time delay for subevent 0 and output pulse length

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EVT0DL	Subevent 0 delay after event condition, in 1uS or system clocks (selected in CFG_DLYC)																														

<event_delay1> Time delay for subevent 1 and output pulse length

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EVT1DL	Subevent 1 delay after event condition, in 1uS or system clocks (selected in CFG_DLYC)																														

STS Event status register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SEARM	SEVT1	SEVT0			SEVT1D	SEVT0D	ECT	Event counter, reports number of detected events since event was enabled, recycled, cleared if event is disabled																							
=1-if event armed	=1 if subevent1 counter is active	=1 if subevent0 counter is active			=1 if subevent1 was issued (auto-cleared after read)	subevent0 was issued (auto-cleared after read)																									

Event configuration field: <event_cfg>

The following configuration bits are defined:

Select source for the event, year/day/hour/minute/seconds/microseconds inclusion/ exclusion, clock source, event pulse length, one-shot/repeat mode and other settings. Detailed descriptions in table below are extracted from the *powerdna.h* provided with your PowerDNA Software Suite examples.

Bit	Name	Description
31	CT650_EVT_CFG_EN	Enable event. Once enabled, event can be triggered by the event source. If STE bit is set, event should be pre-armed by the global start trigger, note that stop trigger that follows start trigger can disarm event if the CT650_EVT_CFG_SPTE bit is set. Single (..._RPT=0) event will auto-clear this bit upon finishing generating event pulse(es).
30-29	CT650_EVT_CFG_RSV	Reserved.
28	CT650_EVT_CFG_SPTE	=1 Use global (layer) stop trigger to "disarm" event
27	CT650_EVT_CFG_EV1IRQ	=1 Generate firmware IRQ based on sub-event 1
26	CT650_EVT_CFG_EV0IRQ	=1 Generate firmware IRQ based on sub-event 0
25	CT650_EVT_CFG_DBL	Number of sub-events, 0-single, 1-dual
24	CT650_EVT_CFG_RPT	Set repeat mode (0-one-time, 1-retriggerable)
23-20	CT650_EVT_CFG_EVTPL(N) CT650_EVT_CFG_EVTPL_1MS=15	Specify event pulse length (set to 60µs by default). Special cases are: EVTPL_1MS= 15 to Use 1ms event pulse length.
19-18	CT650_EVT_CFG_ISRC	Clock source for the internal counter
17	CT650_EVT_CFG_STTE	=1 Use global (layer) start trigger to "arm" event
16-12	CT650_EVT_CFG_IRSRC	Reset source for the internal event period counter same codes as for ESRC field
11	CT650_EVT_CFG_YM	=1 For date/time events - enable compare in new logics
10	CT650_EVT_CFG_DM	=1 For date/time events - enable compare in new logics
9	CT650_EVT_CFG_HM	=1 For date/time events - enable compare in new logics
8	CT650_EVT_CFG_MM	=1 For date/time events - enable compare in new logics
7	CT650_EVT_CFG_SM	=1 For date/time events - enable compare in new logics
6	CT650_EVT_CFG_UM	=1 For date/time events - enable compare in new logics
5	CT650_EVT_CFG_EDGE	Specify the active edge of the event source, for event sources 8-31. Set to 1 for event sources 1-7.
4-0	CT650_EVT_CFG_ESRC_...	Event source mode (0 to 31). See next two tables, below

When <event_cfg> is CT650_EVT_CFG_ESRC or CT650_EVT_CFG_IRSRC, the following event sources or internal counter reset sources (i.e. ESRC/IRSRC) must be ORed with <event_cfg> parameters to set the ESRC/IRSRC source.

NOTE: CT650_EVT_CFG_ESRC_DNAB, CT650_EVT_CFG_ESRC_IPC, CT650_EVT_CFG_ESRC_SBT, and CT650_EVT_CFG_ESRC_BCDT are mutually exclusive in ESRC/IRSRC fields. Use one and only one.

Mode	Name (in *powerdna.h*)	Description
31 8	CT650_EVT_CFG_ESRC_DEVT23 CT650_EVT_CFG_ESRC_DEVT0	Digital event source 23 to Digital event source 0
7	CT650_EVT_CFG_ESRC_RES7	Reserved
6	CT650_EVT_CFG_ESRC_DPLL	Digital PLL which follows time-source
5	CT650_EVT_CFG_ESRC_DNAB	DNA bus condition (reserved for debugging)
4	CT650_EVT_CFG_ESRC_IPC	Internal period counter
3	CT650_EVT_CFG_ESRC_SBT	Straight Binary time mode, event condition is "Straight Binary Input time" >= "SB set time"
2	CT650_EVT_CFG_ESRC_BCDT	BCD time mode, event condition is "BCD Input time" >= "BCD set time". BCD time mode allows creation of the events that will repeat every year, month, day, minute or second - by masking unused parameters in CFG0 register. For events with faster than 1sec repetition rate, can be cascaded or internal counter can be used as an event source
1	CT650_EVT_CFG_ESRC_SWF	Software only, note that event sources 2-31 are ORed with software. For the ESRC field software clock is read from EVT_EMP1 register. For the IRSRC field software clock is read from EVT_EMP0 register
0	CT650_EVT_CFG_ESRC_DIS	No active source - event is in disabled state

When <event_cfg> is 8 to 31, the digital event sources are defined by the following modes (remember to also set CT650_EVT_CFG_EDGE):

Mode	Name	Description
8	CT650_EVT_SEVT00	Event 0/ Subevent 0 (start event)
9	CT650_EVT_SEVT01	Event 0/ Subevent 1 (stop event)
10	CT650_EVT_SEVT10	Event 1/ Subevent 0 (start event)
11	CT650_EVT_SEVT11	Event 1/ Subevent 1 (stop event)
12	CT650_EVT_SEVT20	Event 2/ Subevent 0 (start event)
13	CT650_EVT_SEVT21	Event 2/ Subevent 1 (stop event)
14	CT650_EVT_SEVT30	Event 3/ Subevent 0 (start event)
15	CT650_EVT_SEVT31	Event 3/ Subevent 1 (stop event)
16	CT650_EVT_SYNC0	SYNC bus line 0
17	CT650_EVT_SYNC1	SYNC bus line 1
18	CT650_EVT_SYNC2	SYNC bus line 2
19	CT650_EVT_SYNC3	SYNC bus line 3
20	CT650_EVT_PPS	PPS pulse
21	CT650_EVT_EXTT	External time received and applied
22	CT650_EVT_EINV	External sync lost
23	CT650_EVT_TTL0	External TTL input 0
24	CT650_EVT_TTL1	External TTL input 1
25	CT650_EVT_TTL2	External TTL input 2
26	CT650_EVT_TTL3	External TTL input 3
27	CT650_EVT_TTL4	External TTL input 4
28-31	RSVD	Reserved.

Event mode parameters <event_prm>

This sets parameters that differ depending on the selected ESRC mode:

- when ESRC=EVT_CFG_ESRC_DNAB - DNA bus address and additional trigger parameters
- when ESRC=EVT_CFG_ESRC_IPC - event period counter (divider that selects event period)
- when ESRC=EVT_CFG_ESRC_SBT - straight binary seconds and years
- when ESRC=EVT_CFG_ESRC_BCDT - BCD seconds, minutes, hours and day of the year

ESRC=EVT_CFG_ESRC_SBT			
28-12	CT650_EVT_EMP0_SBS	Straight binary seconds of the day	
11-0	CT650_EVT_EMP0_SBY	Straight binary year	

ESRC=EVT_CFG_ESRC_BCDT			
27-22	CT650_EVT_EMP0_SEC	BCD seconds of the day	
21-16	CT650_EVT_EMP0_MIN	BCD minute of the hour	
15-10	CT650_EVT_EMP0_HR	BCD hour of the day	
9-0	CT650_EVT_EMP0_DOY	BCD day of the year	

ESRC=EVT_CFG_ESRC_IPC			
31-0	CT650_EVT_EMP0_IPC	Event period counter divider DIVIDER=IPC+1	

ESRC=EVT_CFG_ESRC_DPLL			
31-24	RSVD	Reserved	
23-0	MPULSES	For event modules 0 and 1 on IRIG-650 logic 0x10210D6 or newer, this specifies the number of pulses that the DPLL should generate between two “counter reset” pulses provided by the source selected in EVT_CFG_ISRC. The DPLL will constantly monitor the time interval between the reset pulses and adjust its frequency to output exactly <MPULSES> of output pulses that will trigger event; also, in case of unstable clock sources all missing/extra pulses will be accounter for thus allowing the creation of a pulse train locked to the external frequency or 1PPS input source.	

Table 3-1 for shows a graphical representation of <event_cfg>, <event_prm>, <event_val>, and <event_sub0/1_dly> fields.

Event mode parameter field <event_val>

This sets parameters that differ depending on the selected ESRC mode:

- when ESRC=EVT_CFG_ESRC_DNAB - DNA bus data
- when ESRC=EVT_CFG_ESRC_SBT - straight binary day of the year (and SB microseconds)
- when ESRC=EVT_CFG_ESRC_BCDT - microseconds

ESRC=EVT_CFG_ESRC_SBT			
30-20	CT650_EVT_EMP1_SBD	Straight binary day of the year	
19-0	CT650_EVT_EMP1_US	Number of microseconds for the SB and BCD time mode (microseconds are always binary, 0-999999) that are used in event generation	

ESRC=EVT_CFG_ESRC_BCDT			
19-0	CT650_EVT_EMP1_US	Number of microseconds for the SB and BCD time mode (microseconds are always binary, 0-999999) that are used in event generation	

Event delay fields <event_sub0_dly> and <event_sub1_dly> specify event subdelays:

Event source selected in the CT650_EVT_CFG_ESRC acts as a trigger that initiates DLY0/DLY1 counters; these counters provide a programmable delay prior to the generating output pulse and/or interrupt. This behavior may be used to create a sequence of events that are triggered by the same source but should be separated in time by the pre-defined intervals. Two sub-events are provided, thus allowing more flexible handling of the timing. Putting 0 into these registers will result in the event pulse/interrupt delayed by 4-5 clocks from the time of the event source due to the delays in the state machine and edge detector.

Bit	Name	Description	
ESRC=EVT_CFG_ESRC_xxx			
31-0	EVTDL	Event delay in system clocks	

3.2.6.2 Synchronous Programming

Event status can be received by using

```
ret = DqAdv650GetEventStatus(hd, devn, evt_chan, flags,
&event_sts);
```

Valid event channels <evt_chan> are [0..3] should correspond with event channels used in DqAdv650SetEvent() to set up events.

Event status is stored in the following structure:

```
typedef struct {
    uint32 event_sts;    // event status
    uint32 event_tstamp; // event timestamp (debug only)
    uint32 event_ad;     // event address/data (debug only)
} EV650_STS, *pEV650_STS;
```

<event_sts>: the event status register provides information about the current condition of the event module, including number of processed events.

Bit	Name	Description
31	CT650_EVT_STS_SEARM	=1-if event is armed by the start trigger (or start trigger is unused)
30	CT650_EVT_STS_SEVT1	=1-if subevent 1 counter is active
29	CT650_EVT_STS_SEVT0	=1-if subevent 0 counter is active

25	CT650_EVT_STS_SEVT1D	=1-if subevent 1 was issued (auto-cleared after read)
24	CT650_EVT_STS_SEVT0D	=1-if subevent 0 was issued (auto-cleared after read)
23-0	CT650_EVT_STS_ECT	Event counter. Counter is incremented when event start condition is detected and cleared when event is disabled

<event_tstamp> is 32-bit tstamp with 10µs resolution which is stored when an "ESRC_DNAB condition" event happens. Not used in SBT, BCD or other modes, and is primarily intended for debugging.

Synchronous Programming Example 1

The simplest event programming example to receive periodic 1PPS events is:

```
evt_chan = CT650_EVENT_CH0;
flags = DQL_IOCTL650_EVT_EN;
event.event_cfg = CT650_EVT_CFG_EN|CT650_EVT_CFG_EV1IRQ|
                  CT650_EVT_CFG_EV0IRQ|CT650_EVT_CFG_RPT|
                  CT650_EVT_CFG_EDGE|CT650_EVT_PPS;
event.event_prm = 0;
event.event_sub0_dly = 0;
event.event_sub1_dly = 0;
event.event_val = 0;
ret = DqAdv650SetEvents(hd, devn, evt_chan, flags, &event,
&param);
```

and then poll event status with:

```
ret = DqAdv650GetEventStatus(hd, devn, evt_chan, flags,
&event_sts);
printf("EvtSts: AD:%x STS:%x DNABTS:%x\n",
event_sts.event_ad, event_sts.event_sts,
event_sts.event_tstamp);
```

Channel CT650_EVENT_CHD is a special channel which will store events into the layer FIFO, which can be retrieved by calling DqAdv650ReadEventFifo() periodically to make sure that FIFO does not overflow:

```
mode = 0;
rq_size = 200;
ret = DqAdv650ReadEventFifo(hd, devn, mode, rq_size, data,
&ret_size);
if (ret_size > 0) {
  printf("EvtDta: ");
  for (i = 0; i < ret_size; i++) {
           printf("%x ", data[i]);
       }
           printf("\n");
       } else {
           printf(" no events\n");
       }
}
```

To disable an event, call DqAdv650SetEvents() for that channel with a nulled <event> structure (e.g. set all <event> structure fields to 0).

3.2.6.3 Asynchronous Event Setup

Asynchronous events require use of an additional handle that corresponds with a separate UDP port on which event handler will wait for event packets to come.

Use `DqAddIOMPort()` to create an additional handle based on the IOM handle returned from `DqOpenIOM()`:

```
ret = DqAddIOMPort(hd0, &a_handle[i], DQ_UDP_DAQ_PORT_ASYNC,
TIMEOUT_DELAY);
```

Asynchronous events can be selected from the following types:

```
typedef enum {
    EV650_CLEAR = 0x1000,     // clear all events
    EV650_EVENT = 0x101,      // event has happened (any
                              // of the described above)
    EV650_PPS_CLK = 0x102,    // 1PPS pacer clock event
    EV650_TIMERDY = 0x103,    // Time received and processed
                              // (based on FPS rate)
    EV650_GPSRX = 0x104,      // GPS data was received
                              // (reserved)
    EV650_ERROR = 0x110       // multiple errors in TK or TA
} event650_t;
```

To set up asynchronous events on channels `0` thru `3` for `EV650_EVENT`, the event should be configured using `DqAdv650SetEvents()`.

Then, asynchronous events should be set using the following call. Notice that the event is configured with the handle received from `DqAddIOMPort()`.

```
ret = DqAdv650ConfigEvents(a_handle, DEVN, evt_chan, 0,
EV650_PPS_CLK, NULL);
```

Call the same function with `EV650_CLEAR` to disable the event.

For example, to enable event that reports time registers (TREGs) every time that data from input is written to them, the unitization looks as follows:

```
// Use special channels for these events
uint32 evt_chan_rdy = CT650_EVENT_CHTRDY;
uint32 evt_chan_err = CT650_EVENT_CHERR;

// timekeeper to be run from timecode
mode = CT650_TKPPS_TIMECODE;
flags = CT650_TKFLG_AUTOFOLLOW|CT650_TKFLG_USENOMINAL;
ret = DqAdv650ConfigTimekeeper(async_hd, devn, mode,
flags);

input = CT650_IN_AM;           // use AM input for timecode
mode = CT650_IN_DISABLED;       // start in disabled state
ret = DqAdv650SetTimecodeInput(async_hd, devn, mode, input,
pPrmDef, pDataDef);

// Set event to send event packet with when time is decoded
and ready to view (1PPS for IRIG-B)
ret = DqAdv650ConfigEvents(async_hd, DEVN, evt_chan_rdy, 0,
EV650_TIMERDY, NULL);
```

```
// Set error event to send packet once at error condition,
then re-enable error events in the
// error handle to avoid a flood of error notification
packets
ret = DqAdv650ConfigEvents(async_hd, DEVN, evt_chan_err,
DQEVENT_ONCE, EV650_ERROR, &error_mask);
```

Now, create a separate thread to receive events:

```
hThread = RunThread(async_thread, (void*)&a_handle);
```

where `void* async_thread(void* param)` is an actual thread function.

Receive events in the event thread:

```
while(!stop) {
    pEvent = NULL;
    ret = DqCmdReceiveEvent(handle->handle, 0, 1000*1000,
&pEvent, &size);

    if ((ret < 0)&&(ret != DQ_TIMEOUT_ERROR)) {
            printf("ERR: %s\n", DqTranslateError(ret));
    }

    if (ret >= 0) {
        pEv650 = (pEV650_ID)pEvent->data;

        // for this packet do conversion in place
        ntoh_pEv650(pEv650);

        if (print_msg) {
            printf("Event=%x size=%d\n", pEvent->event,
pEv650->size);
        }

        switch(pEvent->event) {
            case EV650_EVENT:
// data contains two uint32: SBS seconds, microseconds from TK
// this is where pacing and other types of events configured in
// DqAdv650SetEvents() should be processed
                break;

            case EV650_PPS_CLK:
// data contains two uint32: SBS seconds, microseconds from TK
// this is a precision once-a-second event
                break;

            case EV650_TIMERDY:
// data contains 16 TREG registers in uint32s
// and two uint32: SBS seconds and microseconds from TK

// If we start receiving time re-enable error messages
                if (reenable_errors)
                    ret = DqAdv650ConfigEvents(handle-
>handle, DEVN, CT650_EVENT_CHERR, DQEVENT_ONCE,
EV650_ERROR, &error_mask);
                reenable_errors = FALSE;
               break;
```

```
            case EV650_ERROR:
// data contains content of four uint32 status registers:
// CT650_STS, CT650_TKSTS, CT650_TDSTS, CT650_SWG_STS
                reenable_errors = TRUE;
                break;

            default:
                // should never execute
                break;
        }
    }
}
```

Since `DqCmdReceiveEvent()` is generic, it does not take care of converting received event from network (big endian) to host (little endian on Intel platform) automatically, so you needs to supply byte-order conversion yourself.

For example:

```
void ntoh_pEv650(pEV650_ID pEv650) {
    uint32 i;

    pEv650->chan = ntohl(pEv650->chan);
    pEv650->evtype = ntohl(pEv650->evtype);
    pEv650->size = ntohl(pEv650->size);
    pEv650->tstamp = ntohl(pEv650->tstamp);
    for (i = 0; i < pEv650->size/sizeof(uint32); i++)
        pEv650->data[i] = ntohl(pEv650->data[i]);
    return;
}
```

Note that all events configured by using `DqAdv650SetEvents()` are received as `EV650_EVENT`.

Also, `error` event is configured to be fired once. If it is received, it means that the input doesn't receive proper timecode or has not synchronized with the input timecode yet. Once a proper time is received, error events are re-enabled (again to be received once upon error). Content of four status registers is returned upon error:

- `CT650_STS` – general status
- `CT650_TKSTS` – timekeeper status
- `CT650_TDSTS` – time decoder status
- `CT650_SWG_STS` – time generator status

Please see meaning of these status bits in *powerdna.h*.

3.2.7 GPS programming

The Fastrax IT500 Series GPS module is enabled upon power-up of IRIG-650 board to give it extra time to acquire satellites.

The GPS module has status and control registers and FIFO-based self-cleaning serial interface which deletes the oldest characters when new data becomes available. Thus when the GPS FIFO is read, it always returns the latest data.

The simplest way to get GPS information is to call `DqAdv650GetGPSStatus()` and `DqAdv650ReadGPS()` in a loop. Serial data is stored into the FIFO by the GPS module every second.

```
while(!stop) {
    ret = DqAdv650GetGPSStatus(hd, devn, mode, &gps_status,
&status, &time, &date);
    printf("GPS status: %x\n", status);

    ret = DqAdv650ReadGPS(hd, devn, mode, rq_size, data,
&ret_size);
    data[ret_size] = 0; data[255] = 0;
    printf("%s", data);

    Sleep(1000);
}
```

3.2.7.1 Retrieving GPS data

The function `DqAdv650GetGPSStatus` returns the status of GPS receiver as:

```
// These bits are returned as gsp_status in
DqAdv650GetGPSStatus()
CT650_GPS_ACC_GPSANTSHRT  =1 if antenna is shorted
CT650_GPS_ACC_GPSANTOK    =1 if antenna detected by the GPS
CT650_GPS_ACC_GPSRXD1     Current value of GPS RXD1 pin
CT650_GPS_ACC_GPSTXD1     Current value of GPS TXD1 pin
CT650_GPS_ACC_GPSRXD0     Current value of GPS RXD0 pin
CT650_GPS_ACC_GPSTXD0     Current value of GPS TXD0 pin
CT650_GPS_ACC_GPSFIXV     Fix valid output from GPS;
                          carries 0.5Hz square wave
                          when GPS fixes it's position
CT650_GPS_ACC_GPSAPPLIED  Time is set to GPS time
CT650_GPS_ACC_SATNUM      number of satellites tracked
CT650_GPS_ACC_ACTIVE      GPS tracking is activated
```

The function `DqAdv650ReadGPS` returns a variety of GPS information:

GGA - Global Positioning System Fix Data
Time, position and fix related data for a GPS receiver

```
$GPGGA,114353.000,6016.3245,N,02458.3270,E,1,10,0.81,35.2,
M,19.5,M,,*50
```

Format:

`$GPGGA,hhmmss.dd,xxmm.dddd,<N|S>,yyymm.dddd,<E|W>,v,ss,d.d,h.h,M,g.g,M,a.a,xxxx*hh<CR><LF>`	UTC time of the fix. hh = hours mm = minutes ss = seconds dd = decimal part of seconds
`hhmmss.dd`	

xxmm.dddd	Latitude coordinate. xx = degrees mm = minutes dddd = decimal part of minutes
<N/S>	Character denoting either N = North or S =South.
yyymm.dddd	Longitude coordinate. yyy = degrees mm = minutes dddd = decimal part of minutes
<E/W>	Character denoting either E = East or W = West.
v	Fix valid indicator 1 = GPS fix (SPS) 2 = DGPS fix 3 = PPS fix 4 = Real Time Kinematic 5 = Float RTK 6 = estimated (dead reckoning) (2.3 feature) 7 = Manual input mode 8 = Simulation mode
ss	Number of satellites used in position fix, 00-12. Notice: Fixed length field of two letters.
d.d	HDOP - Horizontal Dilution Of Precision.
h.h	Altitude (mean-sea-level, geoid)
M	Letter M.
g.g	Difference between the WGS-84 reference ellipsoid surface and the mean-sea-level altitude.
M	Letter M.
a.a	-
xxxx	-

RMC - Recommended Minimum Specific GNSS Data
Time, date, position, course and speed data.

```
$GPRMC,114353.000,A,6016.3245,N,02458.3270,E,0.01,0.00,121
009,,,A*69
```

Format:

$GPRMC,hhmmss.dd,S,xxmm.dddd,<N|S>,yyymm.dddd,<E|W>,s.s,h.h,ddmmyy,d.d, <E|W>,M*hh<CR><LF>	
hhmmss.dd	UTC time of the fix. hh = hours mm = minutes ss = seconds dd = decimal part of seconds
S	Status indicator A = valid V = invalid
xxmm.dddd	Latitude coordinate. xx = degrees mm = minutes dddd = decimal part of minutes
<N|S>	Character denoting either N = North or S = South.
yyymm.dddd	Longitude coordinate yyy = degrees mm = minutes dddd = decimal part of minutes
<E|W>	Character denoting either E = East or W = West.
s.s	Speed in knots.
h.h	Heading
ddmmyy	UTC Date of the fix. dd = day of month mm = month yy = year
d.d	Magnetic variation in degrees, not supported
<E|W>	Letter denoting direction of magnetic variation. Either E = East or W = West. Not supported
M	Mode indicator A=autonomous N=data not valid

GSV - Satellites in view

Number of satellites in view, satellite ID (PRN) numbers, elevation, azimuth, and SNR value. The information for four satellites is a maximum per one message, additional messages up to maximum of eight are sent if needed. The satellites are in PRN number order

Example:

```
$GPGSV,3,1,11,29,68,228,47,30,59,151,47,31,44,284,45,02,38
,062,44*7C
$GPGSV,3,2,11,12,28,130,41,10,14,102,35,05,12,110,35,04,11
,040,34*70
$GPGSV,3,3,11,21,05,196,29,16,05,297,28,13,02,021,30*4E
```

Format:

`$GPGSV,n,m,ss,xx,ee,aaa,cn,…….` `,xx,ee,aaa,cn*hh<CR><LF> n`	Total number of messages, 1 to 9
`m`	Message number, 1 to 9
`ss`	Total number of satellites in view
`xx`	Satellite ID (PRN) number
`ee`	Satellite elevation, degrees 90 max
`aaa`	Satellite azimuth, degrees True, 000 to 359
`ch`	Signal-to-noise ratio (C/No) 00-99 dB-Hz. Value of zero means that the satellite is predicted to be on the visible sky but it isn't being tracked.

GSA - DOP and Active Satellites

GPS receiver operating mode, satellites used in the navigation solution reported by the GGA sentence, and DOP values

Example:

```
$GPGSA,A,3,02,21,30,04,16,05,10,12,31,29,,,1.33,0.81,1.06*
02
```

Format:

`$GPGSA,a,b,xx,xx,xx,xx` `,xx,xx,xx,xx,xx,xx,xx,` `xx,p.p,h.h,v.v*hh<CR><` `LF> a`	Mode: M = Manual, forced to operate in 2D or 3D mode. A = Automatic, allowed to automatically switch 2D/3D.
`b`	Mode: 1 = Fix not available, 2 = 2D, 3 = 3D
`xx`	ID (PRN) numbers of GPS satellites used in solution
`p.p`	PDOP
`h.h`	HDOP
`v.v`	VDOP

The IRIG-650 API provides a simple way to synchronize time with GPS time.

First, set the Time Keeper to be driven from a 1PPS GPS clock (this clock is available regardless of whether the GPS acquired satellites):

`CT650_TKPPS_GPS`

Second, allow handling GPS data by internal interrupt routine. This routine is interrupted upon receiving each GPS NEMA string and keeps track of GPS validity status:

```
ret = DqAdv650EnableGPSTracking(hd, devn, TRUE, &status);
```

Third, make sure that inquiry about current GPS status in the loop to see that antenna is OK.

Fourth, wait for GPS tracking data to become active (also take note that the `CT650_GPS_ACC_GPSFIXV` bit changes each 500ms); and when this flag is set, call `DqAdv650SetGPSTime( )` to force GPS time to be written to the TimeKeeper registers by the interrupt service routine:

```
if (status & CT650_GPS_ACC_ACTIVE) // If GPS reception
{                                  // is active
  printf("GPS is active. Adjust board time to GPS time\n");
  ret = DqAdv650SetGPSTime(hd, devn, flags, &status);
}
```

If time has been applied another GPS status bit will be set:
`CT650_GPS_ACC_GPSAPPLIED`

When it is set the timekeeper time matches GPS satellite time and is clocked by GPS 1PPS pulses with 1µs accuracy.

3.2.7.2 Commanding the GPS unit

The user can also program the GPS unit using NMEA commands:

```
int DAQLIB DqAdv650WriteGPS(int hd, int devn, uint32 mode, int
rq_size, char* data, int* copied)
```

NMEA commands are used to change or query settings of the module.

Command Length:
The maximum length of each packet is restricted to 255 bytes

Commands Contents:

Preamble: One byte character; quoted below.

„$"

NMEA ID: This will identify for the NMEA parser that it is command for MediaTek.

Four byte character string.

“PMTK”

Command Number: Three byte character string.

From “000” to “999”

An identifier used to tell the decoder how to decode the command

DataField: The DataField has variable length depending on the command type.

A comma symbol „,“ must be inserted ahead of each data field to help the decoder process the DataField.

*: 1 byte character.

The star symbol is used to mark the end of DataField.

CHK1, CHK2: Two bytes character string.

CHK1 and CHK2 are the checksum of the data between Preamble and „*“.

CR, LF: Two bytes binary data.

The two bytes are used to identify the end of a command.

Sample Command:

`$PMTK000*32`<CR><LF>

For more information please see the “NMEA manual for Fastrax IT500 Series GPS receivers”.

3.2.8 Calibrating the precision oscillator

The IRIG-650 is factory calibrated using a 10MHz precise Rubidium clock generator. The calibration value is stored inside an EEPROM on the board and is loaded into the calibration DAC every time that the IOM powers up/boots up.

The `DqCmdSetCalibration()` function call can be used to write a 12-bit calibration value directly to the calibration DAC. Mid scale is 0x800.

The user calibration value cannot be stored in EEPROM and needs to be programmed every time when IRIG-650 is configured by the user application.

Please contact UEI technical support for more detail and calibration advice.

3.2.9 Custom PLL frequency generation

The IRIG-650 has one (or three for logics after 0x010210D8) on-board PLL driven from precision temperature and oven controlled oscillator. This PLL can be programmed to generate any frequency from 1Hz to 1MHz to a precise 4-digit value in logics before 0x010210D8. PLL output can be routed to any of TTL Out lines as well as SYNCx lines, to be used as a timebase for other layers. See the “Assigning TTL outputs” on page 31 for details.

To program PLL frequency and duty cycle (from 0 to 1 with 16-bit resolution) use the following function call:

```
int DqAdv650ProgramPLL(int hd, int devn, double duty_cycle,
double frequency, double* f_actual)
```

For logics after 0x010210D8, the PLLs on event modules 0 or 1 described in Section **3.2.6.1** should be used, as they provide superior following of input event sources such as IRIG-B timing sources.

3.2.9.1 Clocking other layers from the IRIG-650

The on-board PLL allow one or more layers to be clocked 20x more reliably and with lower jitter up to 100MHz than the CPU layer’s base clock of 66MHz.

By configuring the custom PLL generation as seen above, and then configuring a SYNC line as seen in Section **3.2.4** for either:

- `CT650_OUT_CFG_SRC_CR` (for logics before 0x010210D8)

- `CT650_OUT_CFG_EVENT`*x* (for logic 0x010210D8 or newer)

and routing those clock lines into your receiving layer(s) with `DqAdvRoute-ClockIn(hd, LAYER_TO_BE_CLOCKED, DQ_EXT_SYNC`*x*`)` and configuring the layer to use this input source instead, it is possible to clock the layer from a IRIG-650 PLL.

Appendix A

A. Accessories

The following cables and break-out boards are available for the IRIG-650 layer.

DNA-CBL-650

IRIG cable providing BNC connections for Clock/IRIG signals and 37-pin connections for other I/O; 2ft long; included with purchase of DNA/DNR-IRIG-650.

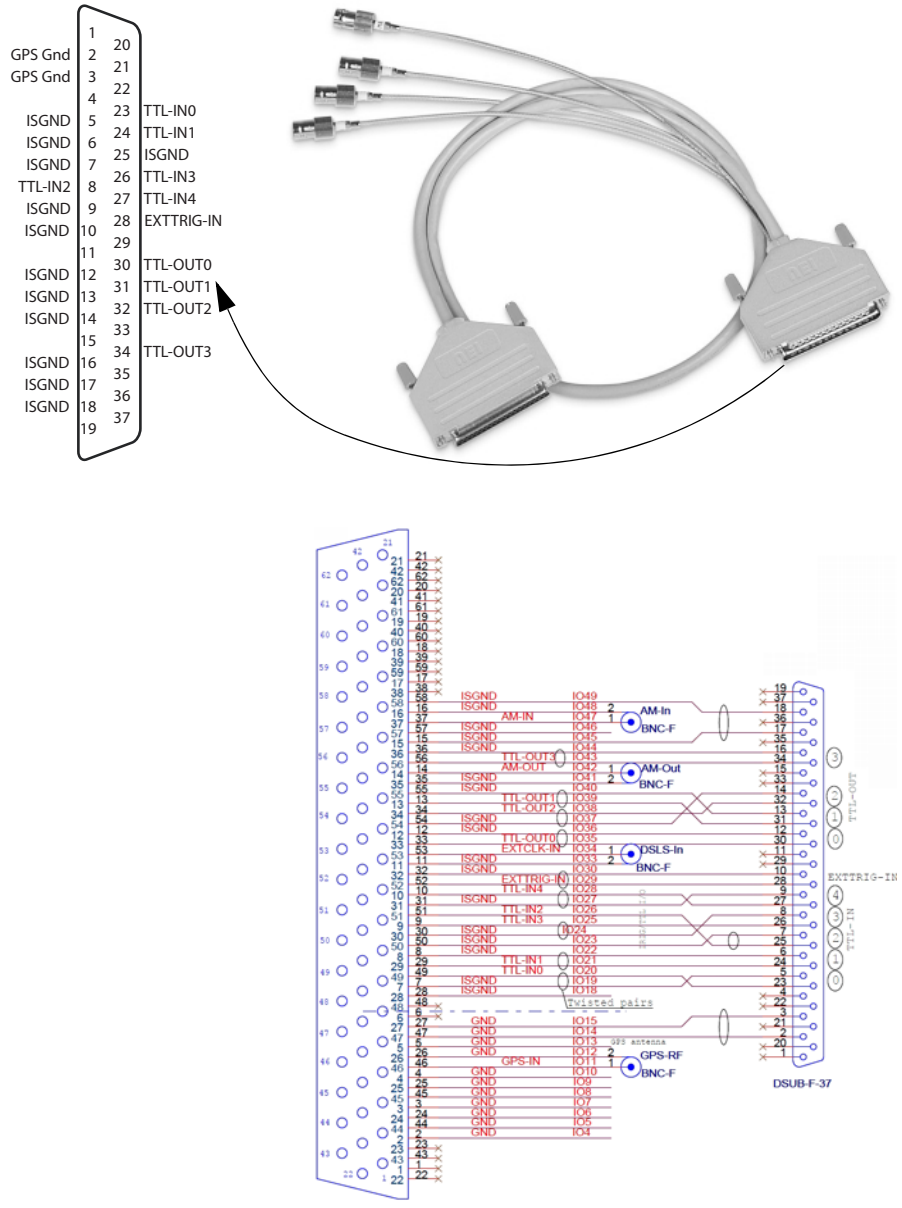


Figure A-1. Pinout, photo, and schema of DNA-CBL-650 accessory

DNA-ACC-650

Break-out board connects to DNA-IRIG-650 primary 62-pin connector and provides GPS, AM-IN, AM-OUT and EXTCLK-IN to mini-BNC connectors.

DNA-BNC-650

Cable for DNA-ACC-650's mini-BNC connector to BNC for external devices; 1ft long.

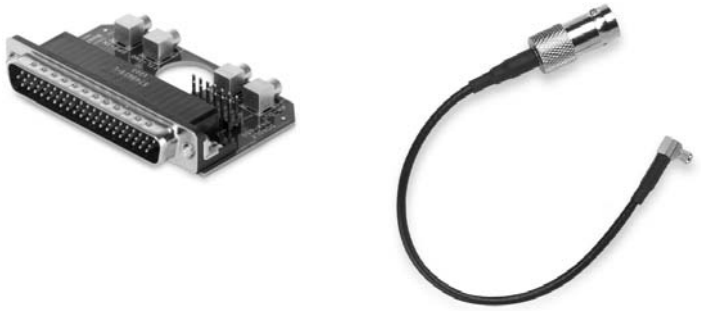


Figure A-2. Photo of DNA-ACC-650 break-out board and BNC-650

Index