82V391x / 8V893xx WAN PLL Device Families – Device Driver User's Guide

Version 1.2

April 29, 2014

Table of Contents

1. Introduction

This document is the programmer's user guide to the device driver for 82V391x / 8V893xx family of devices. The below list describes the supported device families by the device driver.

- 82V391x Device Family
  - 82V3910
  - 82V3911
  - 82V33930
- 8V893xx Device Family
  - 8V89316
  - 8V89317

This user's guide supports device drivers up to **version 1.2**.

All of the information in this document is relevant to any of the supported devices. This document is partitioned into the following sections:

- Software architecture – architecture of device driver
- Getting Started – Installation, initialization, basic tests
- Sample Application – example configurations
- Debug utilities – tools and utilities to diagnose problems

The driver package contains device driver source code which is written in ANSI C (C89). It should be compatible with any standard C compiler and linker.

Included with the device driver package is the following documentation:

- **Release Notes** – Information about the device driver package both current and historical. Any changes to APIs and relevant bug fixes are listed here.
- **Device Driver Application Programmer's Interface (API) Reference Manual** –a comprehensive reference for all functions, data types, and variables.
  - Comes in two formats, PDF and HTML. Both versions support searching
- **Device Driver User's Guide** – this document. How-to guide to the device driver package.

Customers are encouraged to contact IDT support (tsd@idt.com / IDT Technical Support Web Site) for the current list of support devices or any questions regarding the device driver.

2. Software Architecture

2.1. Overview

The device driver is portioned into two parts, the hardware abstraction layer (HAL) and functional modules. The below diagram illustrates a high level architectural view of the driver.

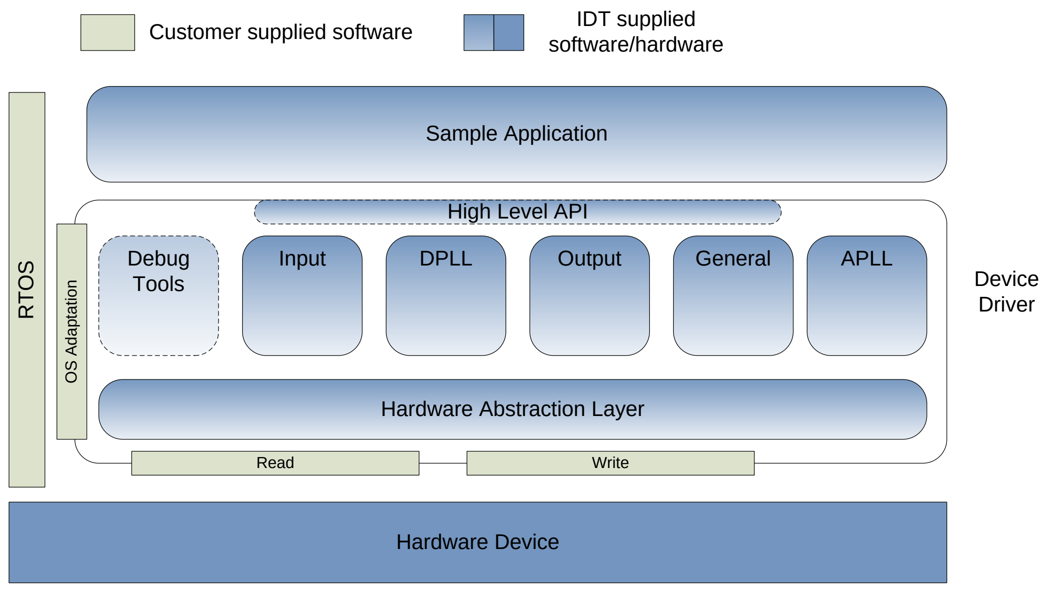

Figure 1 Device driver architecture

2.2. Hardware Abstraction Layer (HAL)

At the lowest layer is the hardware abstraction layer (HAL). This layer of code abstracts specific system access about the targeted device. The HAL acts as a bridge between the device driver and system specific methods to access hardware.

The driver functional modules use the HAL to access the device in a consistent manner. All functional modules use the HAL interface to access the device.

Customers must provide read and write functions to bind to the driver (HAL). Using this architecture a variety of different read/write functions can be "plugged" into the driver. Functional modules remain system agnostic.

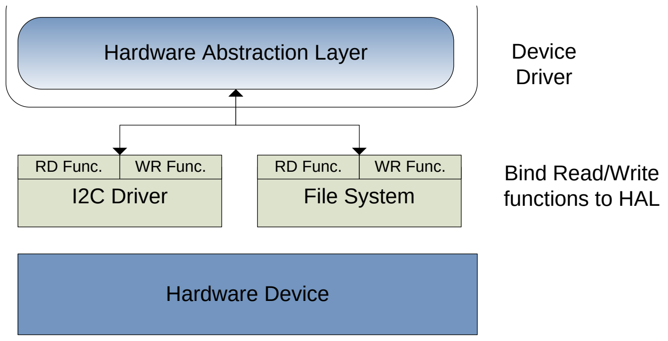

Figure 2 Hardware abstraction layer (HAL)

The driver package contains a number of read/write functions that have been created for the purposes of interfacing the driver to the evaluation platform and the simulated device. They serve as templates for customers to develop their own functions.

2.3. Functional Modules

This section outlines various functional modules present in the device driver. There are 5 core modules and 2 optional modules. For a complete list of application programmer's interfaces please see specific device API Reference Manual.

- **General** – Device wide features
  - Device initialization
  - Interrupt control
  - Device protection mode
- **Input** - Input provisioning and status
  - Input Configuration (including input priority selection)
  - Input Frequency Monitoring
- **Output** – Output provisioning and status
  - Output Configuration (including PPS, 8kHz, 2kHz pulse)
- **DPLL** - All digital and analog phase locked loop (PLL) related functionality is contained in this module.
  - DPLL provisioning
  - Clock path selection (multiple ways to travel through the device)
  - Phase detection
  - DPLL bandwidth and damping factor

- **APLL** – APLL block for generating ultra-low jitter differential outputs
    - APLL provisioning
    - Only required for output path that has APLL populated
- **High Level API** – A set of functions that simplify provisioning of the device. These functions are **optional**.
- **Debug Tools** - This functional module contains a set of tools to aid in debugging issues. An **optional** component to the device driver.

The high level APIs are a set of functions that simplify the provisioning of the device. It uses the core modules (output, general, input, DPLL, and APLL) to achieve this functionality. Customers who have specific requirements can elect not to use these functions and directly call core API functions. In these cases the high level API can be used as additional sample code.

3. Getting Started

As mentioned in the introduction, this section contains the procedure for device driver installation, initialization and basic tests. Follow the below sequence to integrate the device driver into your system.

Procedure for integrating device driver to a customer system is:

1. Decompress the device driver package
2. Perform Operating System Adaptation
3. Implement Hardware Abstraction Layer functions.
4. Modify Makefile to point to build environment (compiler tool chain)
5. Initialize the device driver

The following sub-sections describe the procedure in details.

3.1. Installation

There is no installer for the device driver package. Decompress the package into the desired directory.

Table 1 Unpacking Instructions

Platform	Instructions
Windows	Open package in either • Winrar • Winzip • 7-Zip Or any other decompression utility supporting, zip files.

Unix, Linux, cygwin	Run the following at a prompt:
	`gunzip < foo.tar.gz | tar xvf –`
	or
	`tar xvzf file.tar.gz`

3.2. Directory Structure

The uncompressed 82V3910 driver contains the following directory structure; the driver for other variant (82V3911, 8V89316, etc.) has similar structure:

Table 2 Driver package directory structure

Directory	Description
82V3910/	82V3910 device specific top level (package dependent)
dev/	Device specific source code
sample/	Sample application

82V3911/	82V3911 device specific top level (package dependent). See 82V3910 for directory details.
82V33930/	82V33930 device specific top level (package dependent). See 82V33930 for directory details.
8V89316/	8V89316 device specific top level (package dependent). See 82V3910 for directory details.
8V89317/	8V89317device specific top level (package dependent). See 82V3910 for directory details.
apll/	APLL files top level
access/	Sample HAL routines
include/	APLL header files
src/	APLL source code
test/	APLL unit tests
bin/	Binary file directory
common/	Common files top level
access/	Sample HAL routines
hlapi/	High Level API
include/	Common header files
src/	Common source code
test/	Device driver unit tests (using open-source check utility)
doc/	Documentation directory
driverAPIRefManual/	Device Driver API Reference Manual
html/	HTML version
rtf/	PDF version
driverUserGuide/	Device Driver User's Guide
lib/	Library directory
obj/	Object file directory
osAdaptation/	OS Abstraction
cygwin/	Cygwin sample OS Adaptation files
linux/	Linux x86 sample OS Adaptation files
test/	Device driver unit test
bin/	Device driver unit test binary directory
obj/	Device driver unit test object directory
tools/	Tools directory
annotate/	RGF File annotator
registers/	Register utility to build C header file
rgf/	Register dump utility to load/save register dump (.rgf) file

Below is a select list of important files and their location within the directory structure.

Table 3 Important files

File	Description
idt391x/ReleaseNotes.txt	Release notes
idt391x/license.txt	Software License
idt391x/Makefile	Top level Makefile to generate driver library and sample application
idt391x/82V3910/dev/include/idt82V3910.h ***idt391x/82V3911/dev/include/idt82V3911.h*** ***idt391x/82V33930/dev/include/idt82V33930.h*** ***idt391x/8V89316/dev/include/idt8V89316.h*** ***idt391x/8V89317/dev/include/idt8V89317.h***	Header files to be included by upper (customer/sample) code; device specific
idt391x/82V3910/sample/src/sample.c ***idt391x/82V3911/sample/src/sample.c*** ***idt391x/82V33930/sample/src/sample.c*** ***idt391x/8V89316/sample/src/sample.c*** ***idt391x/8V89317/sample/src/sample.c***	Example code demonstrating sample configurations for device using device driver. Device specific. Contact IDT support to request additional sample configurations.
idt391x/doc/driverAPIRefManual/html/index.html	Starting point for HTML version of API reference manual
idt391x/doc/driverAPIRefManual/rtf/	Location of PDF version to API reference manual
idt391x/doc/driverUserGuide/DriverUserGuide.pdf	Device drive user's guide (this document)
idt391x/osAdaptation/linux/include/osAdapt.h & osNumeric.h	Sample OS Adaptation file for Linux x86
Idt391x/osAdaptation/linux/src/osAdapt.c	
idt391x/osAdaptation/linux/osConfig.mk	Sample build environment makefile for Linux x86
idt391x/common/access	Directory containing sample HAL functions 3 Sets of sample HAL functions provided: • I2C • Simulated (used for unit tests)

3.3. Operating System Adaptation

It is trivial to adapt the device driver to **any** operating system. System integration to the device driver is done through the implementation of macros and typedefs of common data types (char, int, etc.). These macros and typedefs are located in the **osAdaptation** directory. The driver comes with 2 sample OS adaptations:

- Linux x86
- cygwin

The below table lists the required macro implementations (see osAdapt.h for example):

Table 4 OS Adaptation Macros

Macro Name	Description
OS_ASSERT(cond)	Assert on **false** condition. Usually bound to **assert** function in standard C library (assert.h)
OS_ASSERTS(cond,fmt)	Similar to DRV_ASSERT but outputs message **fmt. fmt** is similar argument to printf.
OS_ERROR(fmt)	Unexpected run time error occurred. Message **fmt.** Usually implemented as a log message and an assert.
OS_SETLOGLVL(lvl)	Set highest log level to output. A simple custom function to track highest log level can be written.
OS_GETLOGLVL()	Get highest log level to output.
OS_LOGGER(lvl,fmt)	Log message **fmt** of priority **lvl**. Can be bound to printf function in standard C library (stdio.h) or to existing system logging facilities.
OS_TRACE(fmt)	Log message **fmt** of predefined priority. Output file name, line number and function name. Can be used to trace function calls. Can be bound to printf function in standard C library (stdio.h) or to existing system logging facilities.
OS_MALLOC(numBytes)	Allocate a block of memory sized **numBytes** and return pointer to memory. Usually bound to **malloc** in standard C library (stdlib.h)
OS_FREE(ptr)	Deallocate memory referenced by ptr. Usually bound to **free** in standard C library (stdlib.h)
OS_ZERO(ptr,size)	Zero out block of memory sized **size** referenced by **ptr**. Usually bound to **memset** function in standard C library (string.h).

Common data types are abstracted with typedef, e.g.

```
/** @brief Unsigned char (8-bit) */
typedef unsigned char T_osUint8;
```

For a full list of typedefs, please refer to osAdaptation/osNumeric.h header file.

During the development phase we strongly encourage customers to enable assertions. It will aid in detecting invalid input parameters when using the device driver. For production, assertions can be removed or redirected to a logging system.

3.4. Build Tool Chain Specification

A makefile is provided within the driver package to build a device driver library and sample application. By default it is configured for Linux but can be modified to refer to any build tool chain. The below diagram illustrates the makefile hierarchy.

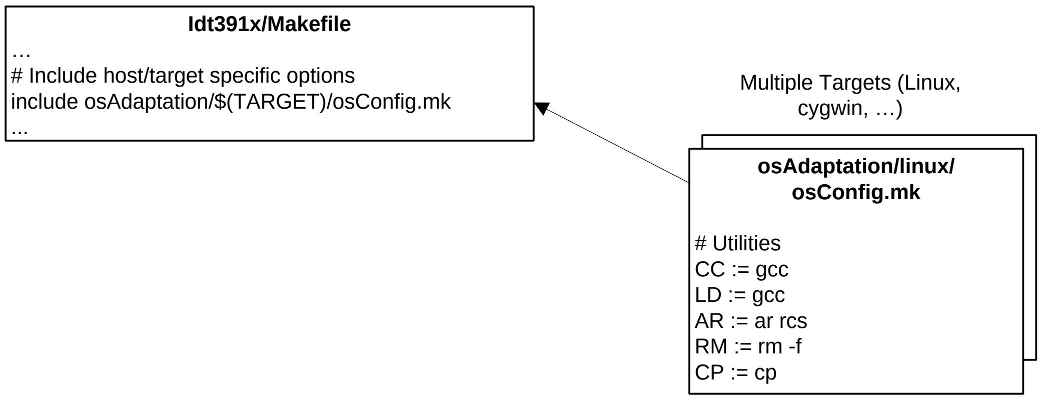
Idt391x/Makefile
…
# Include host/target specific options
include osAdaptation/$(TARGET)/osConfig.mk
...
Multiple Targets (Linux, cygwin, …)
osAdaptation/linux/ osConfig.mk
# Utilities
CC := gcc
LD := gcc
AR := ar rcs
RM := rm -f
CP := cp

Figure 3 Makefile Hierarchy

Customers may specify their own system specific tool chains using this structure.

3.5. Device Access (HAL)

Device access routines are bound to the device driver during driver initialization. The below code snippet illustrates the required functions to bind into the HAL.

```
#include “idt82V3910.h” /* Or any other supported device */

/**
 * \brief Function for reading from device
 * \param usrData optional data passed to write function
 * \param deviceAddr device address (e.g. I2C slave address)
 * \param addrOff address offset
 * \return read data
 */
T_osUint8 myReadFunc (void *usrData, T_osUint8 deviceAddr, T_osUint8 addrOff)
{
   /* System specific code to read a register
    *
    * For example, have I2C controller issue read command
    */
}

/**
 * \brief Function to writing to device
 * \param usrData optional user data passed to read function
 * \param deviceAddr device address (e.g. I2C slave address)
 * \param addrOff address offset
 * \param data data to write
 */
void myWriteFunc (void *usrData, T_osUint8 deviceAddr,
                  T_osUint8 addrOff, T_osUint8 data)
{
   /* System specific code to write to a register
    *
    * For example use I2C controller to initiate a write command
    */
}

/**
 * \brief Function to initialize device access
 *
 * Optional binding
 *
 * \param userData optional user data passed to initialization function
 */
void myInit (void *userData)
{
   /* System specific code to initialize device access mechanism.
    *
    * For example: initialize I2C controller
    */
}

/**
 * \brief Function to deintialize device access
 *
 * Optional binding
 *
 * \param userData optional user data passed to deinitalization function
 */
void myDeinit (void *userData)
{
   /* System specific code to read a register
    *
    * For example: shut down I2C controller
    */
}
```

Figure 4 HAL Bind Functions

3.6. Simulated Device

The device driver comes with a device simulator to enable developers to develop software without requiring a physical device. To setup the device driver to use simulated device specify HAL interfaces located in **common/access/sim** directory. See below initialization example for implementation details.

3.7. Initialization

Once both operating system adaptation and HAL integration functions have been coded, initialization of the driver is trivial.

82V391x / 8V893xx family of devices can be populated with up to two APLLs to produce ultra low jitter on certain output paths. Each APLL supports up to two external crystal connections (XTAL1/3 for APLL1 and XTAL2/4 for APLL2). The supported crystal frequencies are 24.8832MHz (SONET), 25MHz (Ethernet), and 25.78125 MHz (Ethernet FEC). All devices communicate with microprocessor through I2C interface and require an I2C slave address. For detailed APLL population and I2C addressing information, please refer to the corresponding datasheet.

When initializing the driver, user needs to supply APLL external crystal configuration information (ID and frequency) and an unmodified (without direction bit) 7-bit I2C slave address for the device. The APLL external crystal configuration information is defined as following:

```
/**
 */brief Structure containing APLL crystal id and frequency information
 */
typedef struct
{
    T_idtApllXtalId apllXtalId; /**< APLL crystal ID
                                     e.g. APLL_XTAL_1_APLL1 - for APLL1 XTAL1 */
    T_idtApllXtalType apllXtalFreq; /**< APLL crystal frequency
                                         e.g. APLL_XTAL_FREQ_25000KHz –for 25MHz crystal */
} T_idtApllXtal;
```

For the detailed information of each field in the structure, please refer to the "**Device Driver Application Programmer's Interface (API) Reference Manual**".

Below is the sample code on how to initialize the device driver.

```
#include “idt82V3910.h” /* Or any other supported device */
#include “ReadWrite_sim.h” /* For simulated device */
#include “myReadWrite_i2c.h” /* User’s implementation of I2C functions */

/* Assuming the device has the following APLL crystal configuration:
 * - APLL1 XTAL1 at 25MHz
 * - APLL2 XTAL2 at 24.8832MHz
 */
/* APLL XTAL configuration list */
static T_idtApllXtal apllXtalList[] =
{
    /* APLL1 XTAL1 at 25MHz */
    { APLL_XTAL_1_APLL1, APLL_XTAL_FREQ_25000KHz },
    /* APLL2 XTAL2 at 24.8832MHz */
    { APLL_XTAL_2_APLL2, APLL_XTAL_FREQ_24883_DOT_2KHz },
};

/* Number of APLL XTAL configuration */
static T_osUint8 numberOfApllXtal =
       sizeof(apllXtalList)/sizeof(T_idtApllXtal);

/* Assuming i2c slave address is 0x50 */
static T_osUint8 i2cAddr = 0x50;

int main(int argc, char *argv[])
{
  drvHdlr_t   hdlr;
  drvAccess_t deviceAccess;
  int         wantSimulatedDevice = 0;
  /* Bind device access functions */
  if (wantSimulatedDevice) {
    /* Use a simulated device – functions come from ReadWrite_sim.h */
    deviceAccess.initFunc_m    = Init_Sim;
    deviceAccess.deinitFunc_m  = DeInit_Sim;
    deviceAccess.readFunc_m    = RDByte_Sim;
    deviceAccess.writeFunc_m   = WRByte_Sim;
  }
  else {
    /* Access actual device */
    deviceAccess.initFunc_m    = myInit;
    deviceAccess.deinitFunc_m  = myDeinit;
    deviceAccess.readFunc_m    = myReadFunc;
    deviceAccess.writeFunc_m   = myWriteFunc;
  }
  /* No additional user information required. */
  deviceAccess.userData_m    = NULL;
  /* Initialize driver
   *
   *   Driver initialization routine returns a driver handler.
   *   This handler is used to refer to a particular instance.
   */
  hdlr = IDTGeneral_InitDriver("dev82V3910",
                               deviceAccess,
                               IDT_82V3910,
                               apllXtalList,
                               numberOfApllXtal,
                               i2cAddr,
                               1); /* 0-No high level API used
                                    * 1-Can use high level API */
  /* Initialize 3910 device */
  IDTGeneral_InitDevice(hdlr);

  /* Start device usage (i.e. provision device) */

  /* Deinitialize driver – call during system shutdown */
  IDTGeneral_DeInitDriver(hdlr);
  return 0;
}
```

Figure 5 Sample Initialization function

4. Sample Applications

In addition to device driver source code, the package contains sample code for configuring the device. The sample code illustrates a possible design for customer application code. Its primary function is to demonstrate device driver usage. The secondary function is to provide quick provisioning of configurations. Below is an example of sample code.

The last three arguments in "`IDTHlApi_ConfigureOutput`" function in the sample code are APLL related parameters. If the output does not go through APLL, please set them to unused "max" enum value, e.g. for APLL instance id, set it to ""`APLL_INST_MAX`" in case the output does not go through APLL. For detailed description of each function, please refer to the "**Device Driver Application Programmer's Interface (API) Reference Manual**".

```
/**
 * @brief Configure the device in the following configuration:
 *      IN3 (25 MHz) -> T0 +-> OUT1 (25 MHz)
 *                         +-> OUT6 (156.25 MHz)
 * @param hdlr Device driver handler
 */
void IDTSample_ConfigureSampleConfig1(T_idtDrvHdlr hdlr)
{
        /* Disable all outputs */
        IDTHlApi_DisableAllOutputs(hdlr);

        /* Configure input 3, 25MHz */
        IDTHlApi_ConfigureInputFrequency(hdlr, 3, 25000);

        /* Configure DPLL for G8262 Option 1 */
        IDTHlApi_g8262EecOption1(hdlr, DPLL_INSTANCE_1);

        /* Configure output output 1 (25 MHz) using T0 */
        IDTHlApi_ConfigureOutput(hdlr,
                        1,
                        OutFreq_25000k,
                        DPLL_INSTANCE_1,
                        sonetGigeMode_Matching,
                        APLL_CONFIG_MAX,
                        APLL_OUT_SIG_MAX,
                        APLL_CLK_SEL_MAX); /* We're only using one of the DPLLs */


        /* Configure output output 6 (156.25 MHz) using T0 */
        IDTHlApi_ConfigureOutput(hdlr,
                        6,
                        OutFreq_156250k,
                        DPLL_INSTANCE_1,
                        sonetGigeMode_Matching,
                        APLL_CONFIG0, /* Program the APLL configuration 0 */
                        APLL_OUT_SIG_LVPECL,
                        APLL_CLK_SEL_INTERNAL); /* We're only using one of the DPLLs */

        /* Enable output */
        IDTOutput_SetOutputEnable(hdlr, 1, 1);
        IDTOutput_SetOutputEnable(hdlr, 6, 1);
}
```

Figure 6 Sample Code

Sample code is located in [DEVICE]/sample/src directory (where [DEVICE] is 82V3910, 82V3911, 8V89316 ...).

In addition to sample code, the device driver can be compiled to a sample application. This application has the following features:

- Read/Write access of registers
- Execute sample configurations
- Input / Output /DPLL High level API configuration

For usage instructions run the compiled sample application without any parameters.

5. Debug Tools

This final section details various tools provided by the device driver package to aid in debugging issues. Some tools are meant to be used offline. The package currently contains the following tools:

- **Tracing** – Function tracing allows customers to determine which APIs have been called.
- **Save & Restore Registers** – Ability to save and load registers in *.rgf format (compatible with GUI)
- **GUI Register File Annotation** – This utility annotates *.rgf files generated by the GUI. It adds name and bit field names to each register value entry in the file. It makes it easier to understand the provisioned register values.

Further debug tools are planned and will be introduced in the future. We welcome any ideas / feature requests to better provide support for our customers.