DYNAMIC ENGINEERING
150 DuBois St. Suite C Santa Cruz CA 95060
831-457-8891 **Fax** 831-457-4793
http://www.dyneng.com
sales@dyneng.com
Est. 1988

Software User's Guide
(Linux)

Libipack/lib_gen/libipxx

IPACK user libraries
IPACK generic driver

Libipack

Dynamic Engineering
150 DuBois St Suite C
Santa Cruz, CA 95060
831-457-8891
831-457-4793 FAX

Product Description

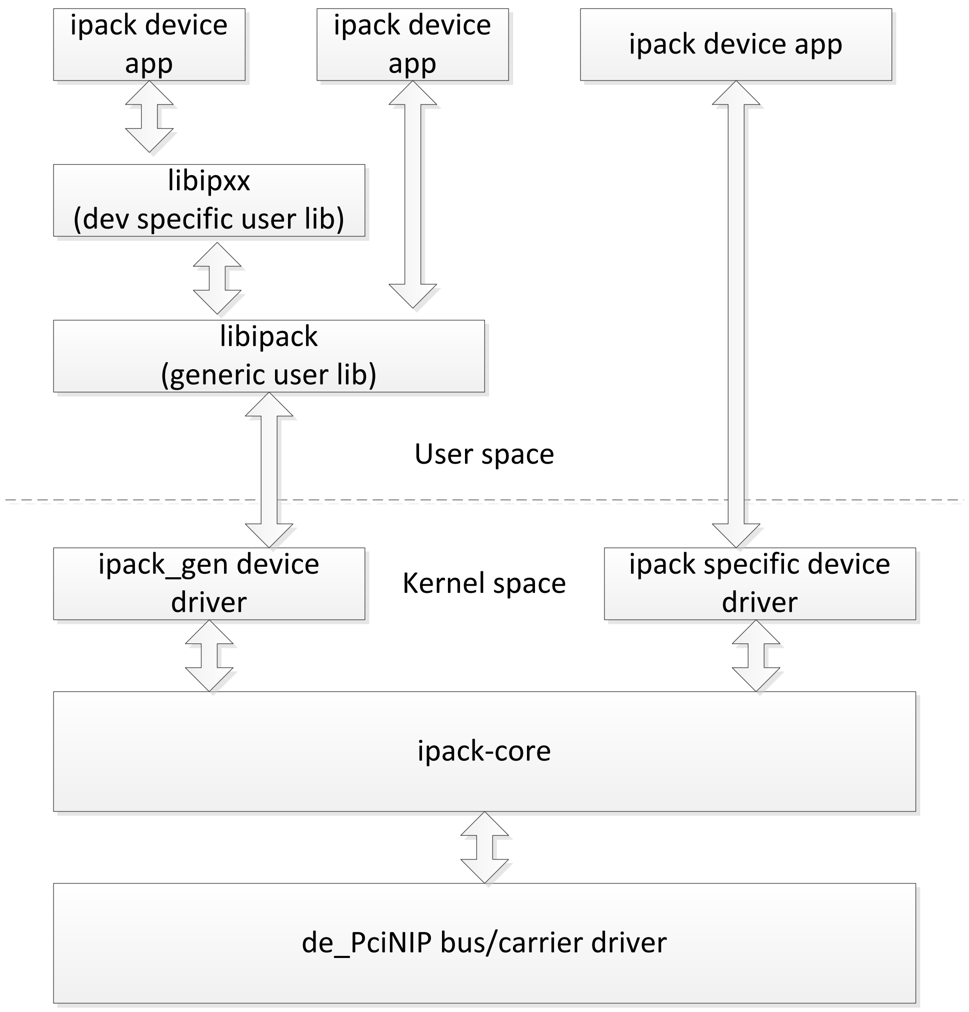
Dynamic Engineering has developed and supplies user-level IPACK (Industry Pack) libraries which support both generic IPACK operations, and device specific functions.
These libraries interface with the ipack-core (Open Source ported from 3.5 kernel) via the ipack_gen(eric) driver.   Thus, this kernel module serves as a gasket between the user-libraries and the ipack-core. The Dynamic Engineering PciNIP driver is a bus/carrier driver supporting all our released carrier/bridge cards interfacing with the ipack-core.

Software Description

As described in the PciNIP SW manual, the ipack-core and de_PciNIP kernel modules must be built and installed prior to utilization of any other IPACK components including those described within this document. Please see that manual for details WRT building and installing these modules.

Based upon specific IPACK device complexity, application developers have multiple methods available for interfacing with the device. A kernel driver may be developed or supplied as depicted on the right side of the diagram below.

This document will address the components and methods depicted on the left side of the diagram. Namely, libipack, lib_gen(eric), and a libipxx library.



The libraries and ipack-gen driver have been validated on an i7 Ubuntu server running 3.8.0-44 kernel (64 bit) SMP (little Endian platform and a P2020 (PPC) target running 3.0.48-rt70 SMP kernel (big Endian platform).

Libipack API descriptions

The following APIs support generic Industry Pack operations and functions. Please review the following descriptions for caveats and general usage details.

```
/*********************************************************************
*  libipack_init
*
*  Initialize library. This function must be invoked prior to utilizing
*  any of the following access routines.  If utilized in conjunction
*  with any Dynamic Engineering module specific libraries, it will be
*  invoked implicitly.
*
*  Parameters:
*  N/A (void)
*
*  Returns:
*  Number of modules upon success, < 0 upon failure
*/
int libipack_init (void);

/*********************************************************************
*  libipack_exit
*
*  Exit/shutdown library. This function should be invoked upon
*  application termination.  If utilized in conjunction with any
*  Dynamic Engineering module specific user libraries, that library
*  will invoke this function upon exit.
*
*  Parameters:
*  N/A (void)
*
*  Returns:
*  0 upon success, < 0 upon failure
*/
int libipack_exit (void);
```

```
/*********************************************************************
*  ipack_get_modules
*
*  Find/get a list of specified IPACK modules. A specific manufacturer
*  ID, model number, and design ID can be specified,
*  or any combination of manufacturer, model number, and design ID via
*  the parameter IPACK_ANY_ID.  A list of IPACK modules meeting the *
*  criteria is returned. If utilized in conjunction with any Dynamic *
*  Engineering module specific user libraries, that library will invoke
*  this function during discovery processing.
*
*  Parameters:
*  man_id       - IPACK manufacturer ID or IPACK_ANY_ID
*  model_num    - IPACK model number or IPACK_ANY_ID
*  design_id    - IPACK driver id (low byte used by Dynamic Engineering
*                 to specify design variant)  or IPACK_ANY_ID.
*  modules      - pointer to an array of size (libipack:MAX_IP_MODULES)
*                 if find_all is true.  Otherwise, an array of a single
*                 element is sufficient.
*  Returns:
*  Number of modules upon success, < 0 upon failure
*/
int ipack_get_modules (unsigned char man_id, unsigned char model_num,
             unsigned char design_id, int find_all, unsigned *modules);


/*********************************************************************
*  ipack_get_modinfo
*
*  Get IPACK module info for specified device.
*
*  Parameters:
*  handle       - IPACK handle returned from ipack_get_modules
*  modinfo      - Module info returned in this structure
*
*  Returns:
*  Number of modules upon success, < 0 upon failure
*/
int ipack_get_modinfo (ipack_handle_t handle,
                                          ipack_modinfo_t* modinfo);
```

```
/*********************************************************************
*  ipack_set_irq_parms
*
*  This function sets required IRQ processing parameters.  It must be
*  invoked prior to enabling a module interrupt.  If utilized in
*  conjunction
*  with any Dynamic Engineering module specific user libraries,
*  that library will invoke this function during configuration
*  processing.
*  The input parameters ip_int_clr* specify irq processing required to
*  clear a module interrupt, set one of more to null if not required
*  to clear interrupt.
*
*  Parameters:
*  handle       - IPACK handle returned from ipack_get_modules
*  vector       - Pointer to vector parameters.
*  read         - Pointer to read parameters.
*  write        - Pointer to write parameters.
*
*  Returns:
*  0 upon success, < 0 upon failure
*/
int ipack_set_irq_parms (ipack_handle_t handle, ip_int_clr_vec_t*
            vector, ip_int_clr_rd_t* read, ip_int_clr_wr_t* write);

/*********************************************************************
*  ipack_wait_irq
*
*  This function awaits interrupt processing completion.
*  ipack_set_irq_parms must be invoked (directly, or indirectly via
*  module specific library prior to invocation.
*
*  Parameters:
*  handle       - IPACK handle returned from ipack_get_modules.
*  vector_rd    - Vector read from int space during last interrupt.
*  data_rd      - Data read during during last interrupt(s)
*  rmw_data_rd  - Data read and written back during last interrupt(s)
*  timeout      - timeout specified in Linux jiffies, 0 or WAIT_FOREVER
*                 are valid values.
*
*  Returns:
*  0 upon success, < 0 upon failure (likely timeout).
*/
int ipack_wait_irq (ipack_handle_t handle, unsigned short *vector_rd,
      unsigned int *data_rd, unsigned int *rmw_data_rd,
      unsigned   timeout);
```

```
/*******************************************************************
*  ipack_readX
*
*  The following read functions reads from the specified memory
*  module region at the specified offset.
*
*  Parameters:
*  handle       - IPACK handle returned from ipack_get_modules.
*  region       - ipack_space_t (IO, ID, MEM, or int space).
*  offset       - byte offset from base of specified region.
*  count        - Number of elements to read.
*  opts         - ipack_rw_opts_t
*                 (IPACK_LO_WD|IPACK_HI_WD|IPACK_AUTO_INC)
*  vals         - Pointer to buffer value(s) read during access.
*                 (count # of values)
*
*  Returns:
*  Number of bytes read upon success, < 0 upon failure.
*
*  Special considerations:
*  ipack_read64 will be defined if and only if the target platform
*  supports quad word reads natively.
*/
int ipack_read8 (ipack_handle_t handle, ipack_space_t region,
    unsigned offset, size_t count, ipack_rw_opts_t opts,
    unsigned char *vals);
int ipack_read16 (ipack_handle_t handle, ipack_space_t region,
    unsigned offset, size_t count, ipack_rw_opts_t opts,
    unsigned short *val);
int ipack_read32 (ipack_handle_t handle, ipack_space_t region,
    unsigned offset, size_t count, ipack_rw_opts_t opts,
    unsigned int *vals);
int ipack_read64 (ipack_handle_t handle, ipack_space_t region,
    unsigned offset, size_t count, ipack_rw_opts_t opts,
    unsigned long *vals);
```

```
/*****************************************************************
*   ipack_writeX
*
*   The following write functions writes to the specified module memory
*   region at the specified offset.
*
*   Parameters:
*   handle        - IPACK handle returned from ipack_get_modules.
*   region        - ipack_space_t (IO, ID, MEM, or INT space).
*   offset        - byte offset from base of specified region
*   count         - Number of elements to written.
*   opts          - ipack_rw_opts_t
*                   (IPACK_LO_WD|IPACK_HI_WD|IPACK_AUTO_INC|IPACK_WR_FLUSH)
*   val           - Pointer to vals to be written (count # of values).
*
*   Returns:
*   Number of bytes written upon success, < 0 upon failure.
*
*   Special considerations:
*   ipack_write64 will be defined if and only if the target platform
*   supports quad word reads natively.
*/
int ipack_write8 (ipack_handle_t handle, ipack_space_t region,
    unsigned offset, size_t count, ipack_rw_opts_t opts,
    unsigned char* vals);
int ipack_write16 (ipack_handle_t handle, ipack_space_t region,
    unsigned offset, size_t count, ipack_rw_opts_t opts,
    unsigned short* vals);
int ipack_write32 (ipack_handle_t handle, ipack_space_t region,
    unsigned offset, size_t count, ipack_rw_opts_t opts,
    unsigned int* vals);
int ipack_write64 (ipack_handle_t handle, ipack_space_t region,
    unsigned offset, size_t count, ipack_rw_opts_t opts,
    unsigned long* vals);
```

Libipxx API descriptions

The following library APIs provide user-level access to specific Dynamic Engineering IPACK modules. Currently, the IP-Parallel-IO, IP-BiSerial-IV-CTRB, IP-BiSerial-IV-SIB IPACK and IP-Optoto-16 modules are supported by a device specific user library. This section shall expand as user libraries are added for other Dynamic Engineering Industry Pack modules.

Libip_pario API descriptions

```
/*****************************************************************
*
*  libip_pario_init
*
*  Initialize library. This function must be invoked prior to
*  utilizing any of the following access routines.  This function
*  returns a list of IP-PARIO modules either containing the first
*  module found, or all modules.
*
*  Parameters:
*  find_all     - (0=find first, 1=find all)
*  design_id    - IPACK driver ID (ip_pario_des_id_t) or
*                 IPACK_ANY_ID
*  modules      - pointer to an array of size
*                 (libpack:MAX_IP_MODULES)
*                 if find_all is true.  Otherwise, an array of a
*                 single element is sufficient.
*  Returns:
*  Number of modules upon success, < 0 upon failure
*/
int libip_pario_init (int find_all, ip_pario_des_id_t design_id,
                                      ipack_handle_t *modules);

/*****************************************************************
*  libip_pario_exit
*
*  Exit/shutdown library. This function should be invoked upon
*  application termination.
*
*  Parameters:
*  N/A, void
*
*  Returns:
*  void
*/
void libip_pario_exit (void);


/*****************************************************************
*  ip_pario_cnfg_io
*
*  Configure IP-PARIO module. This routine is invoked to setup various
*  control parameters for TTL or 485 bits.  If both TTL and 485 modes
*  are supported/utilized by this module, this routine must be invoked
*  twice.
```

```
*
*   Parameters:
*   handle   -   Handle returned in module list (lib_pario_init)
*                specifying which IP-PARIO module to configure.
*   config   -   pointer to IP-PARIO configuration parameters
*                See libip_pario.h for details, and note
*                for ip_pario_bits_t
*
*   Returns:
*   0 upon success, < 0 upon failure
*/
int ip_pario_cnfg_io (ipack_handle_t handle,
                                          ip_pario_conf_io_t* config);

/*********************************************************************
*
*   ip_pario_get_num_nibs
*
*   Get number of nibbles for a module based upon mode.
*
*   Parameters:
*   handle       -   Handle returned in module list (lib_pario_init)
*                    specifying which IP-PARIO modules
*                    which IP-PARIO module
*   mode         -   IP_PARIO_TTL or IP_PARIO_485
*   design_id    -   Pointer to design_id, design_id will be returned in
*                    this parameter, specify NULL if N/A.
*   first_bit    -   Pointer to first I/O bit, first I/O bits will be
*                    returned in this parameter, specify null if N/A.
*
*   Returns:
*   Number of nibbles for this module in specified mode
*
*/
int ip_pario_get_num_nibs (ipack_handle_t handle, ip_pario_mode_t mode,
                        ip_pario_des_id_t* design_id, int* first_bit);


/*********************************************************************
*   ip_pario_set_io
*
*   Set IO bits or other control bits (either TTL or 485) dynamically.
*   ip_pario_cnfg_io should be invoked prior to utilizing this function.
*
*   Parameters:
*   handle   -   Handle returned in module list (lib_pario_init)
*                specifying which IP-PARIO module to configure.
*   mode     -   IP_PARIO_TTL or IP_PARIO_485
*   reg_set  -   Which register set to write (IP_PARIO_CTL0,
*                IP_PARIO_INT_EN0, IP_PARIO_INT_CTL0 or IP_PARIO_POL0.
*   bits  -      Pointer to 3 element array (ip_pario_bits_t).
*                TTL I/O bits are active low, input when set to 1.
*               Bit sense must be set for 485, and direction must be
*               specified when setting bit sense.
*  dir_485  -   Only utilized when setting io_bits in IP_PARIO_CTL0,
*               ignored otherwise, null maybe specified for other
*               register sets.
*  mask     -   Mask specified which bits are written.
*               See libip_pario.h for details, and see note for
*               for ip_pario_bits_t
*
*
*  Returns:
*  0 upon success, < 0 upon failure
*/
int ip_pario_set_io (ipack_handle_t handle, ip_pario_mode_t mode,
    ip_pario_reg_off_t reg_set, ip_pario_bits_t *bits,
    ip_pario_bits_t *dir_485, ip_pario_bits_t *mask);


/*******************************************************************
*
*  ip_pario_get_io
*
*  Read IO bits (either TTL or 485) dynamically and atomically.
*  ip_pario_cnfg_io should be invoked prior to utilizing this function.
*
*  Parameters:
*  handle   -   Handle returned in module list (lib_pario_init)
*               specifying which IP-PARIO module to configure.
*  mode     -   IP_PARIO_TTL or IP_PARIO_485
*  filtered -   Pointer to 3 element array (ip_pario_bits_t).
*               Values returned are after filter application.
*               Specify NULL if don't care.
*  unfiltered-  Pointer to 3 element array (ip_pario_bits_t).
*               Values returned are prior to filter application.
*               Specify NULL if don't care.
*               See libip_pario.h for details, and see note for
*               ip_pario_bits_t
*
*  Returns:
*  0 upon success, < 0 upon failure
*/
int ip_pario_get_io (ipack_handle_t handle, ip_pario_mode_t mode,
              ip_pario_bits_t *filtered, ip_pario_bits_t *unfiltered);

/*******************************************************************
*
*  ip_pario_init_tmrA
*
*  Initiate IP-PARIO timer A.  It configures timer A with the specified
*  duration/period.  An interrupt maybe generated upon expiration
*  and/or generate a square wave on output data bit 23.
*  Interrupt occurence can be determined by invoking
*  ip_pario_await_int.
*
*  Parameters:
*  handle       - Handle returned in module list (libip_pario_init)
*                 specifying which IP-Pario module.
*  int_enbl     - Enable interrupt generation (0=no int, 1= interrupt)
*  wave_enbl    - Square wave enable
*  duration     - 50 usec is minimum duration.
*
*  Special Considerations:
*  The function will fail if timer is currently active, to cancel an
*  outstanding timer, invoke the function ip_pario_reset_tmr.
*  If current value of counter/timer must be read, use timer B, timer A
*  does not support this functionality.
*  Returns:
*  0 upon success, < 0 upon failure
*/
int ip_pario_init_tmrA (ipack_handle_t handle,  uint int_enbl,
                                     uint wave_enbl, uint32_t period);

/*********************************************************************
*
*  ip_pario_init_tmrB
*
*  Initiate IP-PARIO timer B.  It configures timer B with the specified
*  period.  The period generated is guaranteed to be at least as large
*  as specified, and could be twice that specified due to underlying HW
*  implementation.  An interrupt maybe generated upon at the rate of
*  the specified period.  Timer will remain active until canceled by
*  invoking ip_pario_reset_tmr.  Current count can be read via
*  ip_pario_rd_tmrB.
*
*  Parameters:
*  handle   -   Handle returned in module list (libip_pario_init)
*               specifying which IP-PARIO module.
*  int_enbl -   Enable interrupt generation (0=no int, 1= interrupt)
*  period   -   50 usec is minimum duration.
*
*  Special Considerations:
*  The function will fail if a timer is currently active, to cancel an
*  outstanding timer, invoke the function ipctrb_reset_tmr.
*
*  Returns:
*  0 upon success, < 0 upon failure
*/
int ip_pario_init_tmrB (ipack_handle_t handle, uint int_enbl,
                                                  uint duration);
```

```
/*********************************************************************
********
*  ip_pario_rd_tmrB
*
*  Read timer B current counter.  This value is returned in usecs.
*
*  Parameters:
*  handle   -   Handle returned in module list (libip_pario_init) spec-
ifying
*               which IP-PARIO module.
*
*  Returns:
*  Counter offset value on success, < 0 upon failure.
*/
int32_t ip_pario_rd_tmrB (ipack_handle_t handle);


/*********************************************************************
*
*  ip_pario_reset_tmr
*
*  Reset IP-PARIO timer.  This routine will disable/reset specified
*  timer.
*
*  Parameters:
*  handle   -   Handle returned in module list (libip_pario_init)
*               specifying
*               which IP-PARIO module.
*  timer    -   0 = timer A, 1 = timer B
*
*  Returns:
*  0 upon success, < 0 upon failure.
*/
int ip_pario_reset_tmr (ipack_handle_t handle, uint timer);

/*********************************************************************
*
*  ip_pario_await_int
*
*  Await interrupt for the specified PARIO interrupt.  These interrupts
*  types include data bit transitions, timer A, or timer B.
*
*  Parameters:
*  handle       -   Handle returned in module list (libip_pario_init)
*                   specifying which IP-PARIO module.
*  event        -   Interrupt of interest
*  bits_ttl     -   TTL Data bit transitions read upon data interrupt
*                   Specify NULL for timer interrupts or if data bit
*                   transitions are don't care.
*  bits_485     -   485 Data bit transitions read upon data interrupt
*                   Specify NULL for timer interrupts or if data bit
*                   transitions are don't care.
*   timeout       -   Timeout awaiting interrupt in msec.
*
*   Special considerations:
*   Bit transitions/interrupts are reported as a '1' independent of
*   interrupt polarity.
*
*   Returns:
*   0 upon success, < 0 upon failure.
*/
int ip_pario_await_int (ipack_handle_t handle, ip_pario_ints_t event,
   ip_pario_bits_t *bits_ttl, ip_pario_bits_t *bits_485, long timeout);
```

Libipctrb API descriptions

```
/*******************************************************************************
*  libipctrb_init
*
*  Initialize library. This function must be invoked prior to utilizing any
*  of the following access routines.  This function returns a list of IP-CTRB
*  modules either containing the first module found, or all modules.
*
*  Parameters:
*  find_all -   (0=find first, 1=find all)
*  modules  -   pointer to an array of size (libipack:MAX_IP_MODULES)
*               if find_all is true.  Otherwise, an array of a single
*               element is sufficient.
*  Returns:
*  Number of modules upon success, < 0 upon failure
*/
int libipctrb_init (int find_all, unsigned *modules);

/*******************************************************************************
*  libipctrb_init
*
*  Initialize library. This function must be invoked prior to utilizing any
*  of the following access routines.  This function returns a list of IP-CTRB
*  modules either containing the first module found, or all modules.
*
*  Parameters:
*  find_all -   (0=find first, 1=find all)
*  modules  -   pointer to an array of size (libipack:MAX_IP_MODULES)
*               if find_all is true.  Otherwise, an array of a single
*               element is sufficient.
*  Returns:
*  Number of modules upon success, < 0 upon failure
*/
int libipctrb_init (int find_all, unsigned *modules);





/*****************************************************************************
*  libipctrb_exit
*
*  Exit/shutdown library. This function should be invoked upon application
*  termination.
*
*  Parameters:
*  num_modules - Value returned from lib_ipctrb_
*  modules:    - pointer to an array returned from lib_ipctrb_init
*               if find_all is true.  Otherwise, an array of a single
*               element is sufficient.
*  Returns:
*  0 upon success, < 0 upon failure
*/
int libipctrb_exit (unsigned num_modules, unsigned *modules);


/*****************************************************************************
*  ipctrb_initiate_timer
*
*  Start a IP-CTRB module timer. This routine is invoked to initiate a timer.
*  It configures HW to generate an external pulse, and interrupt (if enabled)
*  when the timer expires.  This routine is non-blocking, interrupt occurence
*  can be determined by invoking ipctrb_await_int.
*
*  Parameters:
*  handle   -   Handle returned in module list (lib_ipctrb_init) specifying
*              which IP-CTRB module.
*  channel  -   Channel on the module to initiate a timer (0-7).
*  ext_clk  -   1 = use external clock, 0 = internal.
*  reload   -   0 = normal counter mode, 1 = counter mode/auto reload.
*  int_enbl -   Enable interrupt generation (0=no int, 1= interrupt)
*  duration -   1-4292967295 usec.
*
*  Special Considerations:
*  The function will fail if a timer is currently active, to cancel an
*  outstanding timer, invoke the function ipctrb_reset_chan.  Further,
*  if interrupts are not enabled, timer expiration can be determined by
*  invoking this routine, it will fail with status xx until timer is no longer
*  active (assuming reload==0).
*
*  Returns:
*  0 upon success, < 0 upon failure
*/
int ipctrb_initiate_timer (ipack_handle_t handle, unsigned char channel,
                  unsigned ext_clk, unsigned reload, unsigned int_enbl,
                                             unsigned int duration);


/****************************************************************************
*  ipctrb_initiate_1shot
*
*  Start a IP-CTRB module one shot timer. This routine is invoked to
*  initiate a one shot timer.  It will configure HW to generate an external
*  pulse of the duration specified. The one shot is started based upon an
*  internal or external trigger. It can be configured to start on the rising
*  or falling edge of the selected trigger. This routine is non-blocking,
*  if interrupts are enabled, completion mayb be determined by invoking
*  ipctrb_await_int.
*
*  Parameters:
*  handle   -   Handle returned in module list (lib_ipctrb_init) specifying
*               which IP-CTRB module.
*  channel  -   Channel on the module to initiate the one shot (0-7).
*  ext_clk  -   1 = use external clock, 0 = internal.
*  trigger  -   0=internal trigger (clock), 1=external trigger
*  edge_sel -   0 = falling edge, 1=rising edge of trigger
*  int_enbl -   Enable interrupt generation (0=no int, 1= interrupt)
*  duration -   1-4292967295 pulse width (usec).
*
*  Special Considerations:
*  The function will fail if a one shot is currently active, to cancel an
*  outstanding timer, invoke the function ipctrb_reset_chan.  Further,
*  if interrupts are not enabled, timer expiration can be determined by
*  invoking this routine, it will fail with status xx until one shot is no
*  longer active.

*  Returns:
*  0 upon success, < 0 upon failure
*/
int ipctrb_initiate_1shot (ipack_handle_t handle, unsigned char channel,
                  unsigned ext_clk, unsigned trigger, unsigned edge_sel,
                        unsigned int_enbl, unsigned int duration);

/*****************************************************************************
*  ipctrb_reset_chan
*
*  Reset IP-CTRB channel.
*  This routine will cancel any outstanding request for this channel.
*
*  Parameters:
*  handle   -   Handle returned in module list (lib_ipctrb_init) specifying
*               which IP-CTRB module.
*  channel  -   Channel on the module to reset (0-7).
*
*  Returns:
*  0 upon success, < 0 upon failure.
*/
int ipctrb_reset_chan (ipack_handle_t handle, unsigned char channel);

/*****************************************************************************
*  ipctrb_await_int
*
*  Await timer/counter interrupt on one or multiple channels of of an
*  IP-CTRB module.
*
*  Parameters:
*  handle   -   Handle returned in module list (lib_ipctrb_init) specifying
*               which IP-CTRB module.
*  chan     -   Channel of interest.
*  timeout  -   Timeout awaiting interrupt in msec.
*
*  Returns:
*  0 upon success, < 0 upon failure.
*/
```

```
int ipctrb_await_int (ipack_handle_t handle, unsigned char channel,
                                          unsigned int timeout);
```

Libipsib API descriptions

```
/*****************************************************************************
*  libipsib_init
*
*  Initialize library. This function must be invoked prior to utilizing any
*  of the following access routines.  This function returns a list of IP-SIB
*  modules either containing the first module found, or all modules.
*  Further all channels will be defaulted to SDC/SDT mode configuration.
*
*  Parameters:
*  find_all -   (0=find first, 1=find all)
*  modules  -   pointer to an array of size (libipack:MAX_IP_MODULES)
*               if find_all is true.  Otherwise, an array of a single
*               element is sufficient.
*  Returns:
*  Number of modules upon success, < 0 upon failure
*/
int libipsib_init (int find_all, unsigned *modules);

/*****************************************************************************
*  libipsib_exit
*
*  Exit/shutdown library. This function should be invoked upon application
*  termination.
*
*  Parameters:
*  num_modules - Value returned from libipsib_init.
*  modules:    - pointer to an array returned from lib_ipsib_init
*                if find_all is true.  Otherwise, an array of a single
*                element is sufficient.
*  Returns:
*  0 upon success, < 0 upon error (standard Linux errno)
*/
int libipsib_exit (unsigned num_modules, unsigned *modules);
```

```
/******************************************************************************
*  ipsib_config_ch
*
*  Configure IP-SIB channel
*
*  Parameters:
*  handle   -   Handle returned in module list (libsib_init) specifying
*               which IP-SIB module.
*  channel  -   Channel on the module to initiate a timer (0-1)
*  mode     -   SIB mode of operation (0=SDC/SDT, 1=USIP/USOP)
*  cts_pol  -   Polarity of cts signal (0=active high, 1=active low)
*                 (Don't care for SDC/SDT mode, CTS is disabled).
*
*  Returns:
*  0 upon success, < 0 upon error (standard Linux errno)
*/
int ipsib_config_ch (ipack_handle_t handle, unsigned char channel,
                                  unsigned mode, unsigned cts_pol);

/******************************************************************************
*  ipsib_read
*
*  Read from a SIB channel.
*
*  Parameters:
*  handle   -   Handle returned in module list (lib_ipsib_init) specifying
*               which IP-SIB module.
*  channel  -   Read channel.
*  *buf     -   Buffer of size count.
*  count    -   Number of words to read.
*  timeout  -   Non-zero value implies blocking read of duration timeout
*               msec. If 0 is specified, non-blocking read
*
*  Special Considerations:
*  Channel must be configured prior to initiating a read or write.
*  Maximum read/write size is 511 bytes
*
*  Returns:
```

```
*  Count of words read, < 0 upon error (standard Linux errno)
*/
int ipsib_read (ipack_handle_t handle, unsigned char channel,
             unsigned short *buf, unsigned count, unsigned int timeout);

/*****************************************************************************
*  ipsib_write
*
*  Write to a SIB channel.
*
*  Parameters:
*  handle   -   Handle returned in module list (lib_ipsib_init) specifying
*             which IP-SIB module.
*  channel  -   Write channel.
*  *buf     -   Buffer of size count.
*  count    -   Number of words to write.
*  timeout  -   Non-zero value implies blocking write of duration timeout
*             msec. If 0 is specified, non-blocking write.
*
*  Special Considerations:
*  Channel must be configured prior to initiating a read or write.
*  Maximum read/write size is 511 bytes
*
*  Returns:
*  Count of words written, < 0 upon error (standard Linux errno)
*/
int ipsib_write (ipack_handle_t handle, unsigned char channel,
             unsigned short *buf, unsigned count, unsigned int timeout);

/*****************************************************************************
*  ipsib_reset_chan
*
*  Reset IP-SIB channel.
*  This routine will reset channel without impacting current configuration
*  settings.
*
*  Parameters:
*  handle   -   Handle returned in module list (lib_ipsib_init) specifying
*             which IP-SIB module.
*  channel  -   Channel on the module to reset (0-1).
*
*  Returns:
```

```
*  0 upon success, < 0 upon error (standard Linux errno).
*/
int ipsib_reset_chan (ipack_handle_t handle, unsigned char channel);
```

Libipopto API descriptions

```
/*****************************************************************************
*  libipopto_init
*
*  Initialize library. This function must be invoked prior to utilizing any
*  of the following access routines.  This function returns a list of IP-OPTO
*  modules either containing the first module found, or all modules.
*  The counter (CTB) will be started during initialization.
*
*  Parameters:
*  find_all -   (0=find first, 1=find all)
*  modules  -   pointer to an array of size (libipack:MAX_IP_MODULES)
*               if find_all is true.  Otherwise, an array of a single
*               element is sufficient.
*  Returns:
*  Number of modules upon success, < 0 upon failure
*/
int libipopto_init (int find_all, unsigned *modules);

/*****************************************************************************
*  libipopto_exit
*
*  Exit/shutdown library. This function should be invoked upon application
*  termination.
*
*  Parameters:
*  num_modules - Value returned from lib_ipopto_init
*  modules:    - pointer to an array returned from lib_ipctrb_init
*
*  Returns:
*  0 upon success, < 0 upon failure
*/
int libipopto_exit (unsigned num_modules, unsigned *modules);
```

```
/******************************************************************************
*  ipopto_cfg_wavefm
*
*  Configure and initiate or terminate waveform generation (CTA).  Once
*  waveform is configured and enabled, it can be utilized for FET control
*  and/or interrupt generation.  Interrupt generation and FET switching
*  will occur at the rate of period/2.
*
*  Parameters:
*  handle   -   Handle returned in module list (libipopto_init) specifying
*               which IP-OPTO module.
*  wav_enbl -   Enable/disable waveform generation
*               (IPOPTO_ENBL or IPOPTO_DISABLE)
*  period   -   Period of waveform in usec
*               (Don't care if wav_enbl == IPOPTO_DISABLE);
*
*  Special Considerations:
*  The function will fail if waveform generation is currently active and
*  waveform generation is currently active.  To modify waveform, it first
*  must be idle.
*  ipopto_await_int maybe utilzed to determine interrupt assertion.
*
*  Returns:
*  Actual waveform period (> 0) if successful, < 0 upon failure
*/
int ipopto_cfg_wavefm (ipack_handle_t handle, ipopto_enbl_t wav_enbl,
                                            unsigned int period);

/******************************************************************************
*  ipopto_fet_ctrl
*
*  This function configures 1 or more FET channels.  The channel(s) may
*  operate in manual or waveform driven mode.  Manual mode can
*  enable/disable FET (on/off).
*  In auto mode, FET switching is driven by waveform configured via
*  ipopto_cfg_wavefm.
*
*  Parameters:
*  handle    -  Handle returned in module list (libipopto_init) specifying
*               which IP-OPTO module.
*  chan_msk  -  Bit mask specifying which channels to configure
*               e.g. 0x8001 means configure channels 15 and 0.
*  mode_msk  -  Bit mask specifying mode for specified channels.
*              e.g. 0x0001 = Channel 15 : Manual
                             Chanel   0 : Auto (waveform driven)
*  enbl_msk  -  Bit mask specifying on/off (Don't care for auto mode).
*              e.g. 0x1001 == 0x1000, Channel 0 on.
*
*  Special Considerations:
*  If auto mode specified (waveform driven), ipopto_cfg_wavefm must be invoked
*  prior to this function for successful execution.  Otherwise call will fail.
*
*  Returns:
*  Bit mask status, 0 returned on success, !0 upon failure.
*   e.g. for example above.
*   0x0001 = config failed for channel 0
*/
unsigned short ipopto_fet_ctrl (ipack_handle_t handle, unsigned short chan_msk,
                             unsigned short mode_msk, unsigned short enbl_msk);


/*****************************************************************************
*  ipopto_await_int
*
*  Await timer/counter interrupt from CTA (if waveform generation is enabled).
*  This function will enable interrupt generation when invoked, and disable
*  interrupt generation upon exit.
*
*  Parameters:
*  handle   -   Handle returned in module list (libipopto_init) specifying
*              which IP-OPTO module.
*  timeout  -   Timeout awaiting interrupt in msec.
*
*  Special Considerations:
*  If waveform generation has not been enabled, this function will immediately
*  fail and return an error.
*
*  Returns:
*  0 upon success, < 0 upon failure.
*/
int ipopto_await_int (ipack_handle_t handle, unsigned int timeout);
```

```
/*****************************************************************************
*  ipopto_get_counter
*
*  This function reads the current 32 bit counter value (CTB).  The counter
*  is automatically initiated during initialization.  This value is
*  converted to usec based upon IP BUS speed setting.  This counter can
*  be reset upon read completion via the reset parameter.
*
*  Parameters:
*  handle   -   Handle returned in module list (libipopto_init) specifying
*               which IP-OPTO module.
*  reset    -   IPOPTO_ENABLE (reset) or IPOPTO_DISABLE (don't reset).
*
*  Returns:
*  Counter value in usecs if successful, < 0 upon failure.
*/
int ipopto_get_counter (ipack_handle_t handle, ipopto_enbl_t reset);
```

Installation

1)	Install ipack and de_PCIeNIP kernel modules, see SW manual for the de_PciNIP.
2)	Copy ipack_gen.c, ipack_gen.h (ipack_gen) to your module build directory. Invoke the system “make.” Alternatively a makefile for ipack_gen has been included for out of tree kernel module build. If this build method is utilized, cd to the build directory and invoke the script ./build_all. This script will invoke the Makefile to build ipack_gen.ko, compile/archive both libipack, libiphv, and libipctrb as well as building a test applications (ip_IoApp, ip_TimerApp, and ip_1ShotApp).
3)	Copy the resulting ipack.ko module to the target platform/directory.
4)	Copy the startup script bnm to the target.
5)	Invoke the script (./bnm), it will perform an insmod of ipack_gen and create the required device. The script may be invoked from the systems rc.local file as well.

Sample applications

libip_pario

The applications ip_ParioApp.c and ip_ParioTmr.c demonstrate proper usage of library functions/operations for both libipack and libip_pario. As previously mentioned, the Dynamic Engineering IP-Parallel-IO modules are employed for demonstration purposes.

1)	The build_all script contained in the build sub directory will compile, and archive the libraries, compile the sample apps, as well as invoking Make for the kernel module ipack_gen.ko.

Invocation parameters (ip_ParioApp)

The application can be run either as a single instance (one instance performs reads and writes), or two instances, one reader, one writer demonstrating simultaneous module operation.

Sample application invocation is as follows:

Single instance invocation:
./ip_io mod_num b mode (0=TTL, 1=485
Two instances (two terminals)
./ip_io mod_num r mode

```
./ip_io mod_num w mode
```

The application expects that a loopback fixture is attached to the IP-PARALLEL-IO module(s). It validates proper I/O and interrupt generation for all such modules installed. If the fixture is not attached, the test will fail for that module.

Invocation parameters (ip_ParioTmr)

The application demonstrates proper timer operation

Sample application invocation is as follows:

```
./ip_pario_tmr_mod_num timer_sel (a or b) period (usec)
```

libipctrb

The applications ip_Timer.c and ip_1ShotApp.c demonstrates proper usage of library functions/operations for both libipack and libctrb.

1) The build_all script contained in the build sub directory will compile, and archive the libraries, compile the sample apps, as well as invoking Make for the kernel module ipack_gen.ko.

Invocation parameters (ip_TimerApp, ip1ShotApp)

The applications can be run either standalone or in conjunction with a Dynamic Engineering Test fixture. In standalone mode, only internal clock and triggering can be demonstrated/validated.

The test fixture generates an external clock, and propagates an external trigger pulse. If used to validate external trigger functionality, only 1 channel can be tested at a time. **Further, the value EXT_FIXTURE must be defined (top of source file libipctrb.c) to utilize the external test fixture for external triggers. In normal operation, EXT_FIXTURE must be undefined or #undef EXT_FIXTURE.**

Application invocation is as follows:

ip_TimerApp invocation:

ip_timer mod_num (0=internal|1=external)clock (0=norm|1=reload)mode duration(usec) num_chnls(0-7|8, if 8 specified no need to specify list)channel_list

For example,
ip_timer 0 0 0 1000 1 0

The application will exercise the timer functionality on module 0, channel 0 using the internal clock in normal mode with a timer duration of 1 msec. The application will execute 500,000 iterations by first initiating a timer, then awaits the corresponding completion interrupt. The app will continue until until a failure is detected or interrupted with a <CTRL-C>.

ip_timer 0 0 0 1000 8
Same test as above, except all 8 channels are executed.

ip_1ShotApp invocation:

ip_1Shot mod_num (0=internal,1=external)clock trigger(0=internal|1=external) edgeSel(0=falling|1=rising) num_chnls(1|8) channel

For example,
ip_1Shot 0 0 0 1 1000 1 7

The application will exercise the 1-shot functionality on module 0, channel 7 using the internal clock, internal trigger, rising edge with a pulse width of 1 msec. The application will execute 500,000 iterations by first initiating a 1-shot, then awaits the corresponding completion interrupt. The app will continue until until a failure is detected or interrupted with a <CR> from the shell

Note: 1-shot can be run for all 8 channels as above with the app, however, only 1 port can be run when specifying external trigger, otherwise the app will fail.

libipsib

The application ip_SibApp.c demonstrates proper usage of library functions/operations for both libipack and libipsib. This application can exercise the HW in either USIP/USOP or SDT/SDC mode.

1) The build_all script contained in the build sub directory will compile, and archive the libraries, compile the sample apps, as well as invoking Make for the kernel module ipack_gen.ko.

Invocation parameters (ip_SibApp)

The application can only be executed in conjunction with a Dynamic Engineering Test fixture. One fixture supports SDT/SDC more, the other supports USIP/USOP. Both channels may be exercised simultaneously in USIP/USOP mode. Only 1 channel can be run in SDC/SDT mode

The value EXT_FIXTURE must be defined (top of source file libipsib.c) to utilize the external test fixtures. In normal operation, EXT_FIXTURE must be undefined or #undef EXT_FIXTURE.

Application invocation is as follows:

ip_sib invocation:

Two instances of the application must be executed per channel. One instance is the reader and must be started first. The writer instance must be initiated within 5 seconds:

ip_sib mod_num channel(0|1) reader(1=reader|0=writer) mode(0=SDT|1=USOP/USIP) cts_polarity(0=active high|1=low) [pkt_len (1-511 optional)] [num_iterations optional]

For example,

Reader invocation
ip_sib 0 1 1 0 0

Writer invocation
ip_sib 0 1 0 0 0

In this example, the applications will execute in USIP/USOP mode on module 0, channel 1, CTS active high. Since optional parameters are not specified, default packet length of 256 16 bit words and iteration count of 500000 will be utilized. The reader awaits packet reception and validates the data received. The applications can be terminated early via <CTRL-C>

libipopto

The application ip_OptoApp.c demonstrates proper usage of library functions/operations for both libipack and libipopto. This application can exercise the HW for both manual and waveform driven FET switching.

1) The build_all script contained in the build sub directory will compile, and archive the libraries, compile the sample apps, as well as invoking Make for the kernel module ipack_gen.ko.

Invocation parameters (ip_OptoApp)

The application should be used conjunction with a Dynamic Engineering Test fixture. This test fixture enables visual confirmation of proper FET switching via LEDs populating the fixture.

Application invocation is as follows:

ip_opto invocation:

Manual mode validation:
ip_opto 0 m (assuming carrier is populated with one IP-OPTO module)

You should observe each LED cycle on/off beginning with LED 0 (channel 0) every ½ second. This will continue for 60 iterations or until aborted via <CTRL-C>.

Waveform mode validation:
ip_opto 0 w 500000 (period in usec)
Maximum period is approximately 134 seconds, minimum period is 10 usec.

Two alternate patterns are executed. Every other LED will be lit for both patterns. One pattern begins with LEDs 0,2,4,… being waveform driven Other pattern, LEDs 1,3,5,.. are waveform controlled. LEDs not waveform driven are disabled. Waveform driven LEDs are cycled on/off twice based upon the specified period, then the next pattern is executed. This cycle is repeated 60 iterations or until aborted via <CTRL-C>.

Support Contract

Dynamic Drivers are provided AS-IS and sometimes our clients need a little help. Please refer to the support contract page on our website for options about getting help with your driver use and SW development.

http://www.dyneng.com/TechnicalSupportFromDE.pdf

Warranty and Repair

Please refer to the warranty page on our website for the current warranty offered and options.

http://www.dyneng.com/warranty.html

Service Policy

Before returning a product for repair, verify as well as possible that the suspected unit is at fault. Then call the Customer Service Department for a RETURN MATERIAL AUTHORIZATION (RMA) number. Carefully package the unit, in the original shipping carton if this is available, and ship prepaid and insured with the RMA number clearly written on the outside of the package. Include a return address and the telephone number of a technical contact. For out-of-warranty repairs, a purchase order for repair charges must accompany the return. Dynamic Engineering will not be responsible for damages due to improper packaging of returned items. For service on Dynamic Engineering Products not purchased directly from Dynamic Engineering contact your reseller. Products returned to Dynamic Engineering for repair by other than the original customer will be treated as out-of-warranty.

Out of Warranty Repairs

Software support contracts are available to update, add features, change for different revisions of OS etc. Please contact Dynamic Engineering for these options.

For Service Contact:

Customer Service Department
Dynamic Engineering
150 DuBois St. Suite C Santa Cruz, CA 95060
831-457-8891
InterNet Address support@dyneng.com