RIBFDAQ Cook ES

This document shows how to make a event sender (ES) for RIDFDAQ.

Hidetada Baba (RIKEN)
baba@ribf.riken.jp

March 3, 2009
## Contents

1 Introduction .................................. 3
   1.1 Software package .......................... 3
   1.2 Trigger .................................... 3
   1.3 Installation of babirDAQ ................. 3
       1.3.1 Debug mode ............................. 4

2 Cook Event Sender ............................ 5
   2.1 Communication of DAQ processes .......... 5
   2.2 babies with dummy mode .................. 7
       2.2.1 Run DAQ processes ..................... 7
       2.2.2 Make dummy data ....................... 7
       2.2.3 Setting by DAQ controller ............. 8
       2.2.4 Start test run .......................... 8
       2.2.5 Stop test run ........................... 9
   2.3 Programming of event sender ............. 9
       2.3.1 Command and structure ................. 9
       2.3.2 Messaging method ....................... 10
       2.3.3 Receive start command from event builder . 11
       2.3.4 Connect to event builder ............... 13
       2.3.5 Send data to event builder ............. 14
       2.3.6 Receive stop command from event builder . 14

3 Data format .................................. 15
   3.1 Header rule ................................ 15
   3.2 Class ID ................................... 15
       3.2.1 Global block header .................... 16
       3.2.2 Global block ender ..................... 16
       3.2.3 Global block number .................... 16
       3.2.4 Event data header ....................... 16
       3.2.5 Segment data ............................ 16
       3.2.6 Scaler data ............................... 16
       3.2.7 Status data ............................... 16
       3.2.8 Comment data .............................. 16
Chapter 1

Introduction

1.1 Software package

For RIBFDAQ, there are three software packages of babirlDAQ, NBBQ, and ANAPAW. babirlDAQ has been developed for RIBFDAQ, and it includes:

- Information manager,
- Event builder,
- Event sender,
- DAQ controller,
- Analysis front-end.

NBBQ is a DAQ software package for a small system. It includes:

- Device driver,
- DAQ controller,
- Analysis front-end.

Device driver is used by not only NBBQ but also babirlDAQ. ANAPAW is an analysis software based on PAW. This software is developed by S. Takeuchi.

1.2 Trigger

Dead time (Busy time) should be shared with all front-ends. Typical VETO circuit is shown in Figure 1.1. VETO signal is OR of “Busy signal of each front-end” + “DAQ Start/Stop signal”. While RUN is stopping, the trigger should be inhibited by “DAQ Start/Stop signal”. The pulse from the output register has non negligible width, it should be also input OR circuit of the VETO signal.

1.3 Installation of babirlDAQ

You can install in the following way:

1. su
   Switch to super user.
2. cd /usr
   Installation directory is ’/usr/babirl’.
3. tar zxf /tmp/babirl-date-year.tar.gz
4. ln -s /usr/babirl/babirl-date-year.tar.gz /usr/babirl
5. cd /usr/babirl
6. make clean
7. make

1.3.1 Debug mode
With the compilation option `-DDEBUG`, DAQ processes will run with debug mode. With this mode, you will see many verbose comments. To use this option, edit `/usr/babirl/common.mk`

```cpp
#common.mk (original)
CC = gcc
#CFLAGS = -Wall -O2 -I../include -D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE -DDEBUG -g
CFLAGS = -Wall -O2 -I../include -D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE

#common.mk (debug mode)
CC = gcc
CFLAGS = -Wall -O2 -I../include -D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE -DDEBUG -g
#CFLAGS = -Wall -O2 -I../include -D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE
```

And then, do 'make clean' and 'make'.

Figure 1.1: Typical VETO circuit

![Diagram of a typical VETO circuit]
Chapter 2

Cook Event Sender

2.1 Communication of DAQ processes

Main components of babirlDAQ are babild and babies. babild is an event-builder, babies is an event-sender. Figure 2.1 shows the brief schematic of the data way. babicon is a DAQ controller. babier is internal thread of babild. babirldrv and babirtdrv are the device driver for Linux and RTLinux, respectively. ddrv is a user program for babies with the dummy mode.

To realize the common dead-time for all, the start/stop sequence is strictly defined. Figure 2.2 shows the timing chart of the start/stop sequence. The detail sequence is described below. babissm is a start/stop manager which controls the veto signal for the trigger. babinfo is a information manager which calculate scaler information, mainly. For start sequence:

1. Veto signal for the trigger is high.
2. Put start command to babild from babicon.
3. babild puts start command to each babies, by turns.
4. Each babies puts start command to own driver.
5. Each driver replies start-ack to each babies.
6. Each babies replies start-ack to babild.
7. When babild receives all start-ack from all babies, babild puts start command to babinfo.
8. babild puts start command to babissm.
9. babissm clears the veto signal for the trigger.

For stop sequence:

1. Put stop command to babild from babicon.
2. babicon waits to get stop-ack from babild.
3. babild puts stop command to babissm.
4. babissm set the veto signal for the trigger.
5. babild puts stop command to each babies, by turns.
6. Each babies put stop command to own driver.
7. Each driver returns the last event-fragment-data to each babies.
8. Each babies transfers the last event-fragment-data to babild.
CHAPTER 2. COOK EVENT SENDER

Figure 2.1: The data way of babirlDAQ.

Figure 2.2: The timing char of the start/stop sequence.
2.2 Babies with dummy mode

It can run with dummy mode, then you can check the communication between event sender and event builder. About the installation of babirlDAQ, please refer Sec. 1.3.

2.2.1 Run DAQ processes

You can run DAQ processes by following commands (super user privilege is required):

- Event-builder computer
  1. /usr/babirl/babinfo/babinfo
  2. /usr/babirl/babild/babild -l 1

  The last argument of babild is EFN. This number will be stored into event-built data.

- Event-sender computer
  1. /usr/babirl/babies/babies -d 2

  The last argument of babies is EFN. This number is important to communicate with the event builder.

To stop DAQ process, please use:

- Event-builder computer
  1. kill -2 'cat /var/run/babild'
  2. kill -2 'cat /var/run/babinfo'

- Event-sender computer
  1. kill -2 'cat /var/run/babies'

2.2.2 Make dummy data

You can make dummy raw data file on event-sender computer:

- /usr/babirl/devtool/mkdummryidf 2 50 100 2 32 /tmp/dummy.ridf

The synopsis of ‘mdummryidf’ command is:

<table>
<thead>
<tr>
<th>mdkdummryidf EFN EVTSIZE BLKN SCRID SCRN OUTFILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFN : Event fragment number</td>
</tr>
<tr>
<td>EVTSIZE : 1 event size (in short word)</td>
</tr>
<tr>
<td>BLKN : Number of creating blocks</td>
</tr>
<tr>
<td>SCRID : Scaler ID</td>
</tr>
<tr>
<td>SCRN : Number of scaler channel</td>
</tr>
</tbody>
</table>
2.2.3 Setting by DAQ controller

From event-builder computer or event-sender computer, run `babicon`.

1. `/usr/babirl/babicon/babicon EBHOSTNAME`  
   EBHOSTNAME is a hostname of event-builder computer or IP address. With this command, you can connect event builder.

2. `BABICON> getconfig`  
   Show event builder configuration. If you find some list in ‘Event fragment’, please delete. For example, if you see:

<table>
<thead>
<tr>
<th>Event fragment</th>
<th>ID</th>
<th>Hostname</th>
<th>Nickname</th>
<th>on/off</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11</td>
<td>172.27.224.169</td>
<td>rtl</td>
<td>(on)</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>172.27.224.45</td>
<td>scr</td>
<td>(scr)</td>
</tr>
</tbody>
</table>

3. `BABICON> seteflist 11 del`  
   With these two commands, the list of event fragment will be deleted. Then, create new event fragment (sender) information.

4. `BABICON> seteflist 12 del`  
   With these two commands, the list of event fragment will be deleted.

5. `BABICON> seteflist 2 add ESHOSTNAME NICKNAME`  
   ESHOSTNAME is the hostname or IP address of event-sender computer. NICKNAME is just a nickname.

6. `BABICON> getesconfig 2`  
   You can see parameters of event-sender (`babies`).

7. `setesconfig 2 host EBHOSTNAME`  
   With this command, `babies` will send data to EBHOSTNAME.

8. `BABICON> esconnect 2`  
   `BABICON> esdisconnect 2`  
   `BABICON> getesconfig 2`  
   You will be able to find ‘connect : 0’.

2.2.4 Start test run

You can start test run with no-save mode by `babicon`:

1. `BABICON> nssta`  
   With this command, run will be started.

2. `BABICON> getevtnumber`  
   This command shows the event-built number. Now, you will see ‘No event build’.
   If run is starting, at event-sender computer, you can put dummy data to `babies`.

1. `/usr/babirl/devtool/ddrv 2 /tmp/dummy.ridf`  
   This `ddrv` command put data to `babies` from file. The number of 2 is EFN.

2. **ENTER**  
   With ENTER key, you can send next block data.
2.3 Programming of Event Sender

You can confirm event-built number by `babicon`.

1. BABICON> geteventnumber
   To store data into file:
   1. sethdlist 0 path /tmp/
   2. sethdlist 0 on
   3. setrunname testdata
      Stored file name will be ‘testdata****.ridf’.
   4. setrunnumber 0
      Next runnumber will be 1.
   5. BABICON> wth
      Write header.
   6. BABICON> start
   7. BABICON> stop
      Ender will be required.

With these commands, raw data will be stored into /tmp/testdata0001.ridf. To check raw data, you can use:

```
| chkridf /tmp/testdata0001.ridf
```

### 2.2.5 Stop test run

To stop run:

1. BABICON> stop
   `babicon` will be blocked.

To complete the stop sequence, please terminate `ddrv` process running on event-sender computer.

1. [Ctrl-C]

Then, run will be stopped. With this dummy mode, you will find these line at the stop sequence:

```
<table>
<thead>
<tr>
<th>Last event number</th>
</tr>
</thead>
</table>
| EFN Nickname | EVTNR  
| 2 nickname | 0 |
```

EVTN is the event number counted by event sender. With this dummy mode, event sender do not count the event number. The ‘getevtnumber’ command of `babicon` shows event-built number in event builder.

### 2.3 Programming of event sender

#### 2.3.1 Command and structure

Here, commands and some useful structures for event sender are listed.

```c
#define BABIRL_COM_SIZE 1024*10 ///< Buffer size for babirl tcp commands

/** 17601 : for receiver */
#define ERRCVPORT 17601
/** 17651-18000 : for babies communication port */
```
## 2.3.2 Messaging method

Messaging and data transfer is done with TCP/IP protocol except for on-line analysis. Basic protocol is:

<table>
<thead>
<tr>
<th>Send</th>
<th>Length (4 Bytes) + Command (4 Bytes) + Content (X Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return</td>
<td>Length (4 Bytes) + Content (X Bytes)</td>
</tr>
</tbody>
</table>

‘Length’ is the byte count of ‘Command’ + ‘Content’ for sending, ‘Content’ for returning. It does not include sizeof ‘Length’. To send ‘ES_SET_CONFIG’ command which sets the configuration of an event sender:

```c
int com, len;
struct stefrc efrc;
char buff[1024];

com = ES_SET_CONFIG;
len = sizeof(com) + sizeof(efrc);
send(sock, (char *)&len, sizeof(len), 0);
send(sock, (char *)&com, sizeof(com), 0);
send(sock, (char *)&efrc, sizeof(efrc), 0);
```
2.3. PROGRAMMING OF EVENT SENDER

recv(sock, (char *)&len, sizeof(len), MSG_WAITALL);
recv(sock, buff, len, MSG_WAITALL);

2.3.3 Receive start command from event builder

When run is started, \textit{babild} puts start command to event senders. The point is that do not return start-ack until VME/VXI is ready to accept the trigger. Receiving and replying sequence is as follow:

Main loop:
/* Global variables */
int efsock, comfd;
struct stefrc efrc;

memset((char *)&efrc, 0, sizeof(efrc));

efrc.efid = EFN; // Event fragment number;
strcpy(efrc.erhost, "EBHOSTNAME"); // Hostname of event builder
efrc.runnumber = -1;
efrc.erport = ERRCVPORT;
efrc.comport = ESCOMPORT + efrc.efid;

/* Make command port */
if(!(comfd = mktcpsock(ESCOMPORT + efrc.efid))) quit();

while(1){
    commain();
}

int commain(void){
    char buff[BABIRL_COM_SIZE],
    int len, com, clen, sock;
    struct sockaddr_in caddr;

    clen = sizeof(caddr);
    if((sock = accept(comfd, (struct sockaddr *)&caddr, (socklen_t *)&clen)) < 0){
        return 0;
    }

    memset(buff, 0, sizeof(buff));
    recv(sock, (char *)&len, sizeof(len), MSG_WAITALL);
    recv(sock, buff, len, MSG_WAITALL);
    memcpy((char *)&com, buff, sizeof(com));

    switch(com){
    case ES_RUN_START:
        memcpy((char *)&arg, buff+sizeof(com), sizeof(arg));
        if(arg == ES_EF_OFF){
            break; /* noop */
    }
}
/* Startup function for ADCs should be done here */
// open_drv();

ret = 1; // Start-ack
len = sizeof(ret);
send(sock, (char *)&len, sizeof(len), 0);
send(sock, (char *)&ret, len, 0);
break;
.
.
.
}

return 1;
}

/** Make TCP server socket (receiver), automatically bind and listen.
 * Return socket number.
 * @param port port number should be bound
 * @return socket number
 */
int mktcpsock(unsigned short port){
  int sock = 0;
  int sockopt = 1;
  struct sockaddr_in saddr;

  memset((char *)&saddr,0,sizeof(saddr));
  if((sock = socket(PF_INET,SOCK_STREAM,0)) < 0){
    perror("bi-tcp.mktcpsock: Can't make socket.\n");
    return 0;
  }
  setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
             &sockopt, sizeof(sockopt));
  saddr.sin_family = AF_INET;
  saddr.sin_addr.s_addr = INADDR_ANY;
  saddr.sin_port = htons(port);
  if(bind(sock,(struct sockaddr *)&saddr,sizeof(saddr)) < 0){
    perror("bi-tcp.mktcpsock: Can't bind socket.\n");
    return 0;
  }
  if(listen(sock,100) < 0){
    perror("bi-tcp.mktcpsock: Can't listen socket.");
    return 0;
  }

  return sock;
}
2.3. PROGRAMMING OF EVENT SENDER

2.3.4 Connect to event builder

Before send event-fragment-data to babild, TCP connection have to be established between event sender and babier.

```c
int efr_connect(void)
{
    int ret;
    ret = 0;
    if(!efrc.connect){
        /* Make data port */
        if(!((efsock = mktcpsend(efrc.erhost, efrc.erport)))){
            return 0;
        }
        send(efsock, (char *)&efrc.efid, sizeof(efrc.efid), 0);
        recv(efsock, (char *)&ret, sizeof(ret), MSG_WAITALL);
        efrc.connect = 1;
    }else{
        return 0;
    }
    return 1;
}

/** Make TCP client socket (sender), automatically connect.
 * Return socket number.
 * @param host server hostname
 * @param port port number should be connected
 * @return socket number
 */
int mktcpsend(char *host, unsigned short port){
    int sock = 0;
    struct hostent *hp;
    struct sockaddr_in saddr;
    if((sock = socket(AF_INET,SOCK_STREAM,0)) < 0){
        perror("bi-tcp.mktcpsend: Can't make socket.
        return 0;
    }
    memset((char *)&saddr,0,sizeof(saddr));
    if((hp = gethostbyname(host)) == NULL){
        perror("bi-tcp.mktcpsend: No such host");
        return 0;
    }
    memcpy(&saddr.sin_addr,hp->h_addr,hp->h_length);
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(port);
    if(connect(sock,(struct sockaddr *)&saddr,sizeof(saddr)) < 0){
        perror("bi-tcp.mktcpsend: Error in tcp connect.
    }
    return sock;
}
```
2.3.5 Send data to event builder

It is simple to send event-fragment-data.

```c
char *buff;
int len;

send(efsock, (char *)&len, sizeof(len), 0);
send(efsock, (buff, len, 0);
```

2.3.6 Receive stop command from event builder

Do not return stop-ack until VME/VXI finishes stop functions.

```c
int commain(void){

    switch(com){
    case ES_RUN_START:
        ret = 1; // Pre-stop-ack
        len = sizeof(ret);
        send(sock, (char *)&len, sizeof(len), 0);
        send(sock, (char *)&ret, len, 0);

        /* Here, do stop functions for ADCs */
        // daq_stop()

        /* send last (residual) data */
        send(efsock, (char *)&len, sizeof(len), 0);
        send(efsock, (buff, len, 0);

        /* send stop-ack */
        len = -1; // len=-1 means stop-ack
        send(efsock, (char *)&len, sizeof(len), 0);

        break;
    }
```
Chapter 3

Data format

The data format in RIBFDAQ is RIDF (RIBF Data Format). Current version of RIDF is 1.3.

3.1 Header rule

In RIDF, all blocks have header word as follow:

<table>
<thead>
<tr>
<th>2bit</th>
<th>2bit</th>
<th>6bit</th>
<th>22bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Ly</td>
<td>Class ID</td>
<td>Size</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Size</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>32bit</td>
</tr>
</tbody>
</table>

- Revision (R)
  Revision bit, 0 = Version 1.x

- Layer (Ly)
  The layer depth of this block. This layer depth is not important.

- Class ID
  Definition of this block. Data format of this block is determined by Class ID.

- Size
  Block size of this block including this header. Unit is short word (22 bit = 4 M). Then maximum size of 1 block is 8 MB. However, for programming reason, it is recommended to use less than 128 kB = 65536 short word.

- Address
  Address indicates that who made this block. It is equal to Event Fragment Number (EFN). The bit length of address is 32 bits, but for now, lower 8 bits are used only. So then, address (EFN) should be 0–255.

For example, 1 block data is constructed as shown in Figure. 3.1.

3.2 Class ID

There are some kinds of Class ID (Table. 3.1).
### Figure 3.1: An example of 1 block structure.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td>Address</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td></td>
<td>Size</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
<td>Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Event Number</td>
<td>Size</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td></td>
<td>Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Segment ID</td>
<td>Size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Segment Data</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
<td>Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Status ID</td>
<td>Size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Status Data</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td></td>
<td>Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Size of this block</td>
<td></td>
</tr>
</tbody>
</table>

3.2.1 Global block header
3.2.2 Global block ender
3.2.3 Global block number
3.2.4 Event data header
3.2.5 Segment data
3.2.6 Scaler data
3.2.7 Status data
3.2.8 Comment data
### Table 3.1: The definition of Class ID

<table>
<thead>
<tr>
<th>Class ID</th>
<th>Group</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Global block header</td>
<td>Event fragment data</td>
</tr>
<tr>
<td>1</td>
<td>Global block header</td>
<td>Event assembly data</td>
</tr>
<tr>
<td>2</td>
<td>Global block header</td>
<td>Event assembly fragment data</td>
</tr>
<tr>
<td>3</td>
<td>Event data header</td>
<td>Event data</td>
</tr>
<tr>
<td>4</td>
<td>Segment data header</td>
<td>Segment data</td>
</tr>
<tr>
<td>5</td>
<td>Comment data header</td>
<td>Comment data</td>
</tr>
<tr>
<td>8</td>
<td>Global block number</td>
<td>Global block number</td>
</tr>
<tr>
<td>9</td>
<td>Global block ender</td>
<td>Global block ender</td>
</tr>
<tr>
<td>11</td>
<td>Scaler data header</td>
<td>Scaler non-clear 24bit</td>
</tr>
<tr>
<td>12</td>
<td>Scaler data header</td>
<td>Scaler clear every buffer</td>
</tr>
<tr>
<td>13</td>
<td>Scaler data header</td>
<td>Scaler non-clear 32bit</td>
</tr>
<tr>
<td>21</td>
<td>Status data header</td>
<td>Status data</td>
</tr>
</tbody>
</table>