NAME

picontrol_ioctl − ioctls for piControl driver

SYNOPSIS

#include <piControl.h>

int ioctl(int *fd***, int** *cmd***, ...);**

DESCRIPTION

The **ioctl**(2) call for piControl accepts many possible command arguments. Most require a third argument, of varying type, here called *argp* or *arg*.

Get information from the configuration file

The web application PiCtory is used to configure the I/O- and virtual modules. It writes the configuration to the file */etc/revpi/config.rsc*. The piControl driver reads this file when it is started. Most applications can use the following commands to retrieve some information from configuration. Then they do not need to parse the configuration file themself.

**KB_GET_DEVICE_INFO_LIST SDeviceInfo ***argp*

This call provides information about the configured and connected modules. A pointer to an array of 64 entries of type *SDeviceInfo* must be passed as argument.
The return value is the number of entries filled the array.

**KB_GET_DEVICE_INFO SDeviceInfo ***argp*

This call provides information about one module. A pointer to a structure of type *SDeviceInfo* must be passed as argument. Ether the element *i8uAddress* must be set to the address of the module which information should be returned. Or the element *i16uModuleType* must be set to the type of the module. In this case it it only possible to retireve informations about the first module of that typ in the confiugration. The available types are defined in with the defines *KUNBUS_FW_DESCR_TYP_...* in the file *common_define.h*.

The struct *SDeviceInfo* used by this ioctls is defined as

```
typedef struct SDeviceInfoStr {
  uint8_t   i8uAddress;          // Address of module in current configuration
  uint32_t  i32uSerialnumber;    // serial number of module
  uint16_t  i16uModuleType;      // Type identifier of module
  uint16_t  i16uHW_Revision;     // hardware revision
  uint16_t  i16uSW_Major;        // major software version
  uint16_t  i16uSW_Minor;        // minor software version
  uint32_t  i32uSVN_Revision;    // svn revision of software
  uint16_t  i16uInputLength;     // length in bytes of all input values together
  uint16_t  i16uOutputLength;    // length in bytes of all output values together
  uint16_t  i16uConfigLength;    // length in bytes of all config values together
  uint16_t  i16uBaseOffset;      // offset in process image
  uint16_t  i16uInputOffset;     // offset in process image of first input byte
  uint16_t  i16uOutputOffset;    // offset in process image of first output byte
  uint16_t  i16uConfigOffset;    // offset in process image of first config byte
  uint16_t  i16uFirstEntry;      // index of entry
  uint16_t  i16uEntries;         // number of entries in process image
  uint8_t   i8uModuleState;      // fieldbus state of piGate Module
  uint8_t   i8uActive;           // == 0 means that the module is not present and no data is available
  uint8_t   i8uReserve[30];      // space for future extensions without changing the size of the struct
} SDeviceInfo;
```

**KB_FIND_VARIABLE SPIVariable ****argp*

Find a variable in the process image by its name. A pointer to a structure of type *SPIVariable* must be passed as argument. Before the call the name of the variable must be written as zero terminated string to the element *strVarName*. After a successful call *i16uAddress* is set to the offset of the variable in the process image. *i16uLength* if the length of the value in bits. It can be 1, 8, 16 or 32. If it is one, *i8uBit* tells the bit position [0..7] in the byte. For other values of *i16uLength* , *i8uBit* is not used. 2 and 4 byte values are stored in little endian byte order. The address is not alligned to a mutiple of the variable length.

The struct *SPIVariable* used by this ioctl is defined as

```
typedef struct SPIVariableStr
{
  char      strVarName[32]; // Variable name
  uint16_t  i16uAddress;    // Address of the byte in the process image
  uint8_t   i8uBit;         // 0-7 bit position, >= 8 whole byte
  uint16_t  i16uLength;     // length of the variable in bits.
                            // Possible values are 1, 8, 16 and 32
} SPIVariable;
```

Set and get values of the process image

**KB_GET_VALUE SPIValue ****argp*

Read one bit or one byte from the process image. Before the call the elements *i16uAddress* and *i8uBit* must be set to the address of the value. This call is more efficient than the usual calls of seek and read because only one function call is necessary.

**KB_SET_VALUE SPIValue ****argp*

Write one bit or one byte to the process image. Before the call the elements *i16uAddress* and *i8uBit* must be set to the address of the value. *i8uValue* must set to the value to write. This call is more efficient than the usual calls of seek and write because only one function call is necessary. If more than on application are writing bits in one output byte, this call is the only safe way to set a bit without overwriting the other bits because this call is doning a read-modify-write-cycle.

The struct *SPIValue* used by this ioctl is defined as

```
typedef struct SPIValueStr
{
  uint16_t  i16uAddress;  // Address of the byte in the process image
  uint8_t   i8uBit;       // 0-7 bit position, >= 8 whole byte
  uint8_t   i8uValue;     // Value: 0/1 for bit access, whole byte otherwise
} SPIValue;
```

**KB_SET_EXPORTED_OUTPUTS const void ****argp*

Write all output values to the hardware at once.

This call is used by the main application controlling the outputs. The application must have a complete copy of the process image. It sets the output values in its own copy and calls this ioctl with a pointer to the image as arguments. This call locks the process image and copies all output values where the **export** checkmark is set in PiCtory. Afterwards piControl transfers the values to the I/O modules. In the current version only one application should call this ioctl.

**KB_DIO_RESET_COUNTER SDIOResetCounter ****argp*

Reset counters and encoders to 0 in an input module.

Inputs of DIO and DI modules can be configured as counters or encoders in PiCtroy. After a reset they start at 0. This call can be used set one or more of the values to 0 at the same time.

The argument must be pointer of a structure of type *SDIOResetCounter*. The element *i8uAddress* must be set to the address of DIO or DI module as shown in PiCtory, e. g. 32 if the DIO is on the right of the RevPi.
The element *i16uBitfield* defines which counters will be reset. If a counter is configured on input I_3 and an encoder is configured on the inputs I_6 and I_7, the bits 2 and 5 must be set to 1, i16uBitfield must be set to the value 0x0024.

The struct *SDIOResetCounter* used by this ioctl is defined as

```
typedef struct SDIOResetCounterStr
{
        uint8_t    i8uAddress;        // Address of module in current configuration
        uint16_t   i16uBitfield;      // bitfield, if bit n is 1, reset counter/encoder on input n
} SDIOResetCounter;
```

KB_SET_EXPORTED_OUTPUTS const void *_argp_

Write exported output values to the real outputs.
The main control application (e.g. a SoftPLC) running on the RevPi can use this function to set all output values at one time. It is very similar to write, but only the values for which the export checkmark is set in PiCtory are set. This allows other applications to control other outputs. The argument pointer must point to a copy of the complete process image. The typical usage is to read the complete process image with read(), change to output values for this cycle and write the marked output values back to the process image.

```
unsigned char PI[4096];
 ...
read(fd, PI, 4096);
/* change PI */
ioctl(fd, KB_SET_EXPORTED_OUTPUTS, PI);
```

KB_SET_OUTPUT_WATCHDOG unsigned int *_argp_

Activate an application watchdog.
The argument is a pointer to the watchdog period in milliseconds. After setting this period value, the write function must be called in shorter periods for this file handle. If it is called within the period, all output value are set to 0 in the piControl driver.
The watchdog can be deactivated by setting the period to 0 or closing the file handle.

Driver Control

KB_WAIT_FOR_EVENT int *_argp_

Wait for a event from the piControl driver.
This is a blocking call. It waits until an event occurs in the piControl driver. The number of the event is writte to the arument pointer.
At the moment there is only a reset event. It is sent to all waiting applications, if a client calls the ioctl **KB_RESET**. The application has to stop its execution and update the offsets of the variables in the process image. Here is a small example:

```
int event;
while (1)
{
  // get variable offsets from piControl
  // start application
  do {
```

```
    if (ioctl(fd, KB_WAIT_FOR_EVENT, &event) < 0)
      exit(-1);
  } while (event != KB_EVENT_RESET);
  //stop application
}
```

KB_RESET **void**

Stop the communication with the I/O modules, reset to all of them, scan for the connected modules, read the configuration file created with **PiCtory**, initialize the modules with the new configuration and restart the communication. All counters are set to 0 and all the outputs to their default values as defined in **PiCtory**.

KB_STOP_IO **int ***argp*

Stop or start the I/O communication.

This ioctl stops, starts or toggles the update of I/Os. If the I/O updates are stopped, piControls writes 0 to the outputs instead of the values from the process image. The input values are not written to the process images. The I/O communication is runnging as normal. On the update of DIO, DI, DO, AIO, Gate modules and the RevPi itself is stopped. There is no change in the handling of virtual modules. The function can used for simulation of I/Os. A simulation application can be started additionally to the other control and application processes. It stops the I/O update and simulates the hardware by setting and reading the values in the process image. The application does not notice this.

The argument is a pointer to an integer variable.

If stop has the value 0, the communication is started.

If it is 1, the communicatio is stopped.

2 will toggle the current mode.

The return value is the new mode, 0 for running, 1 for stopped, <0 in case of an error.

KB_UPDATE_DEVICE_FIRMWARE **unsigned int***arg*

Update firmware

KUNBUS provides **.fwu** files with new firmware for RevPi I/O and RevPi Gate modules. These are provided in the debian paket revpi-firmware. Use *sudo apt-get install revpi-firmware* to get the latest firmware files. Afterwards you can update the firmware with this ioctl call. Unforunatelly old modules hang or block the piBridge communication, if a modules is updated. Therefore the update is only possible when only one module is connected to the RevPi. The module must be on the right side of the RevPi Core and on the left side of the RevPi Connect. This ioctl reads the version number from the module and compares it to the lastet available firmware file. If a new firmware is available, it is flashed to the module.

The argument is the address of module to update. If it is 0, the module to update will be selected automatically.

KB_GET_LAST_MESSAGE **char ***argp*

Get a message from the last ioctl call.

Check, if the last ioctl() call produced a message and copy it to the buffer.

```
char cMsg[REV_PI_ERROR_MSG_LEN];

if (ioctl(PiControlHandle_g, KB_GET_LAST_MESSAGE, cMsg) == 0 && cMsg[0])
  puts(cMsg);
```

SEE ALSO

ioctl(2)

COLOPHON

A description of the project and further information can be found at https://revolution.kunbus.de/forum/