JControl/PLUI

JAVA PROGRAMMABLE POWER-LINE USER INTERFACE WITH BACKLIGHTED 128X64 PIXEL GRAPHIC LCD, JOYSTICK SWITCH, RTC, 256K FLASH MEMORY, BUZZER, KONNEX PL132 COMPLIANT POWER-LINE-BCU, RS232-INTERFACE AND SOFT REALTIME SUPPORT

- **Virtual Machine Core**
  - 8 Bit JAVA™ bytecode execution engine
  - 16 Bit processing word length
  - Max. 255 constant pool entries
  - 2.5k JAVA heap memory
  - 2 MIPS native core speed
  - Automatic garbage collection
  - Multi-threading support with extensions for soft realtime execution
- **Display**
  - 128x64 pixel blue backlighted graphic LCD
  - FSTN technology
  - Viewing area: 46.0 x 23.0 mm
  - Dot pitch: 0.36 mm
  - Backlight and contrast adjustable by software
- **Joystick Switch**
  - 5 switching positions
  - 4 directions (up, down, left, right) and center push
- **Flash Memory**
  - 4 banks of 64k Flash
  - 256 byte sectors
  - > 10,000 erase/write cycles
- **Real-Time-Clock (RTC)**
  - Philips PCF8563 type
  - 32.768kHz crystal oscillator
  - 0.25µA backup current @V_{CC}=3.0V
  - Backup-power-supply by gold-cap
- **Power-Line-BCU**
  - Konnex PL132 compliant power-line BCU, based on ST7537HS1
  - 2400bps, half duplex
  - output signal max. 116dB(µV)
  - receiving signal gain 70dB
  - Capacitive signal coupling to AC network
  - Separated switch-mode power-supply
  - Separate 8052-compliant microcontroller executes protocol stack
  - Internal FT1.2 protocol

- **Power-Supply**
  - 230V AC power supply
  - Euro power cord for power supply and power-line communication
- **Buzzer (Piezo Ceramic)**
  - Controlled by PWM output
  - Frequency range 250Hz..20kHz
- **RS232**
  - 5-Wire RS232-Interface
  - 9-pin Sub-D-connector
  - 11 different baud rates from 300 up to 250.000bps
  - None, even or odd parity
  - Automatic flow control by XON/XOFF or RTS/CTS
- **I/O-Pins (available internally)**
  - 8-pin Molex micro connector
  - 6 Digital General-Purpose I/O Pins
  - 2 I/Os usable as PWM outputs
  - 4 I/Os usable as Analog inputs
- **Physical Dimensions**
  - Size: 125x67x42mm (case w/o connectors)
  - Weight: 320g

DIMENSIONS AND CONNECTORS

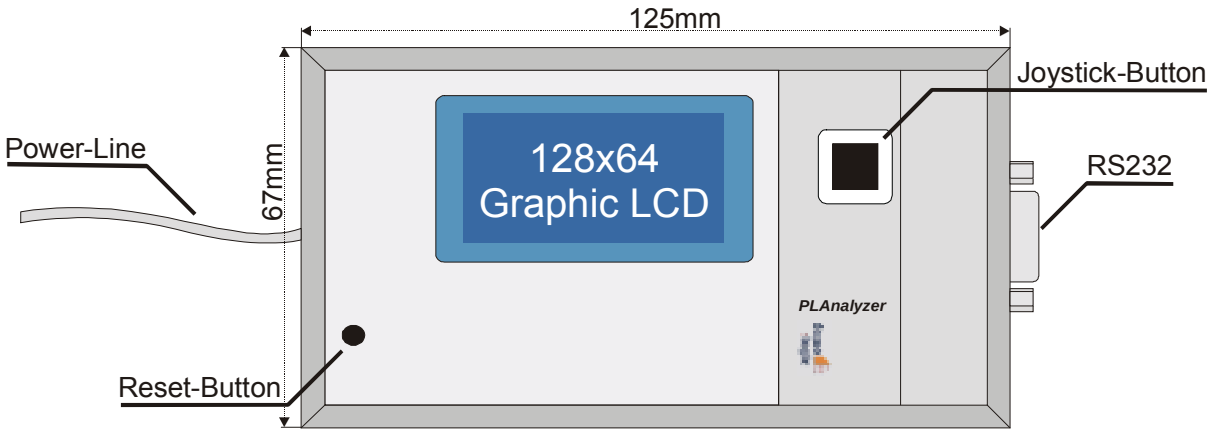


Fig. 1: Dimensions and Connectors of the JControl/PLUI

DEVICE VARIANTS

Sales Type		Native Core Speed	Serial Baud Rates	Flash Organization
JControl/PLUI		2 MIPS	300-250.000bps	256x256x4

Table 1: Different Variants of the JControl/PLUI

GENERAL DESCRIPTION

The JControl/PLUI is a member of the JControl device family, designed as free programmable Konnex PL132-compliant Power-Line-BCU (BCU stands for *Bus Coupling Unit*). The application software will be written in JAVA, compiled and executed by the JCVM8 8 Bit JAVA™ bytecode execution engine. The core runs with 2 MIPS native speed, providing 16 Bit processing word length, 2.5k JAVA heap memory, automatic garbage collection and multi-threading software execution. Applications in the fields of control, measurement and automation are supported by specific extensions for soft-realtime processing.

Fig. 2 gives an overview of the JControl/PLUI. The upper part of the picture shows the JControl System based on the JVCM8 with LCD, Joystick Switch, RTC, Flash Memory, Buzzer and RS232-Interface. The bottom part shows the Power-Line-BCU, combining an opto-galvanically separated interface, an 8052 microcontroller for the protocol stack, a ST7537HS1 power-line-modem, a capacitive power-line interface and a switch mode power-supply.

The JCVM8 offers a set of built-in classes, providing fundamental support of the JAVA programming language and access to all local peripheral components like LCD, joystick switch, RTC, Flash memory, buzzer, communication ports (RS232 and Power-Line-BCU) and some internal universal I/O pins. Extended support is given by class libraries, linked automatically to the application by the JControl/DevelopmentSuite. This mechanism saves memory space, because exclusively the required classes are loaded to the system.

Application programs are loaded via a serial communication interface to the Flash memory, organized into four banks of 64kByte. All banks

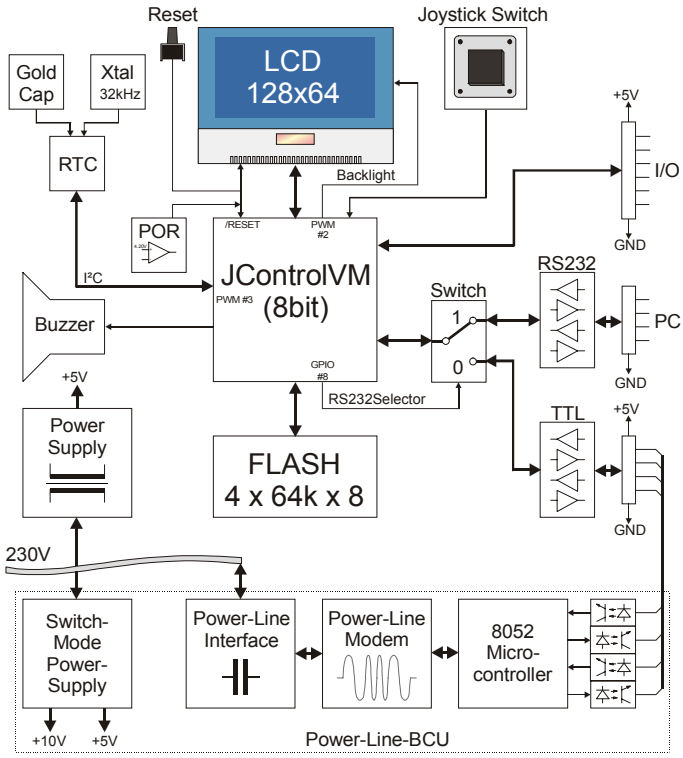


Fig. 2: JControl/PLUI Block Diagram

may be used for application software or non-volatile data.

Various information about the specific JControl device and its current state is available by accessing the *system properties*. In download mode, system properties may be read or written by remote using the *JControl Download Protocol*. Under normal operating conditions, the system properties can be accessed by application software using the methods `getProperty` and `setProperty` of the built-in class `jcontrol.system.Management`.

POWER SUPPLY AND SYSTEM RESET

The JControl/PLUI is powered by the mains (230V), using the supply line for both powering the JControl System and the Power-Line-BCU as well as for coupling the power-line signals into the network. The device operates with 5V DC internally. Under normal operating conditions, the power dissipation of the whole JControl/PLUI is below 1.4W.

The device has no on/off switch and starts immediately by attaching the mains power. When the device is not powered, the internal RTC (Real Time Clock) is backed up by a gold cap.

To ensure a reliable start up phase, a power-on reset generator (POR) holds the reset signal of the VM and LCD while the output voltage of the internal JControl power supply is below 4.2V. After the initialization sequence of the JCVM8 has been executed, the application stored in the first Flash bank (bank 0) will be started (sole exception: see chapter Real Time Clock (RTC)). In the standard configuration of the JControl/PLUI, this is the application software (e.g. Power-Line Analyzer).

FLASH MEMORY ORGANIZATION

The JControl/PLUI offers 4 banks of 64k Flash memory for application software or non-volatile data, numbered as bank 0 to bank 3 (*Fig. 3*). All banks are organized as 256 sectors by 256 bytes, numbered from sector 0 to sector 255. The Flash memory's organization can be detected automatically by reading the system property `flash.format`. The returned string comprises the parameters `<number of sectors>x<bytes per sector>x<number of banks>` (e.g. "256x256x4" for the JControl/PLUI device).

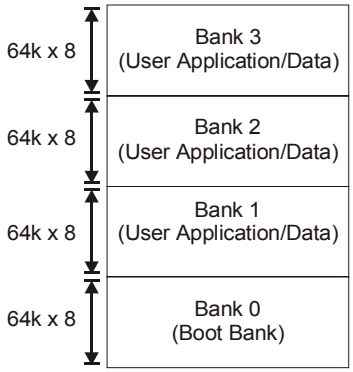


Fig. 3: The two Flash Banks of JControl/PLUI

One Flash bank can be selected for application software execution at a time. The current bank may be changed by restarting the JCVM8 using a different bank (see chapter *Power Supply and System Reset*). After system initialization, always the application software in bank 0 is executed.

The Flash memory can be used to store non-volatile data using the built-in class `jcontrol.io.Flash`. It provides methods to read and write complete sectors in any bank of the Flash memory.

Fig. 4 gives an overview of bank 0's internal structure: Application software is written upwards, starting at sector 0, and non-volatile data is stored downwards, starting at sector 254. This procedure reduces the possibility of resource conflicts between application and data. To offer a linear ascending number of sectors (starting at sector 0) to the application, the class `jcontrol.io.Flash` maps access to the logical sector 0 to the physical sector 254 of the Flash memory, access to logical sector 1 to the physical sector 253 and so on. The uppermost sector of bank 0 (sector 255) is used to store non-volatile system properties.

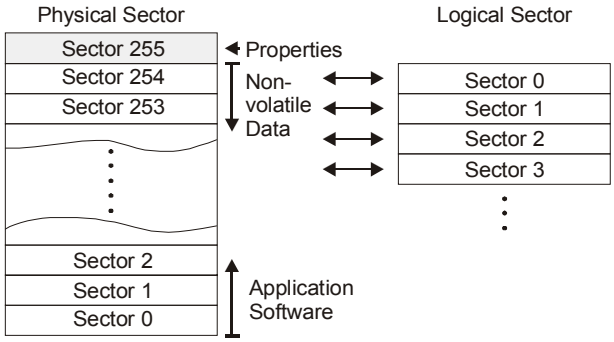


Fig. 4: Internal Structure of Flash Bank 0

For applications that makes use of the flash memory independently of its architecture, the external class `jcontrol.storage.FlashStream` is provided. It represents a memory cached data stream for reading and writing continuous data to or from the non-volatile flash memory.

DOWNLOAD MODE

The *system download mode* is a fundamental functionality of the JCVM8, implemented in every JControl device. It is used for uploading and downloading application software or data from a host computer to the Flash memory, for auto identification of the JControl device and for reading or writing system properties by remote. The download mode is used e.g. by the development tools like *JCManager* and *PropertyEdit*.

The system download mode may be entered by one of following four cases:

(1) Directly after the initialization sequence of the JCVM8: If no valid application software is available in bank 0 of the Flash memory, the device enters the system download mode.

(2) During normal operating conditions: If the virtual machine is restarted by the method `switchBank()` of the built-in class `jcontrol.system.Management` and the new Flash bank contains no valid application software.

(3) The system download mode may be enforced by pushing the joystick switch down while resetting the device as shown in *Fig. 5*.

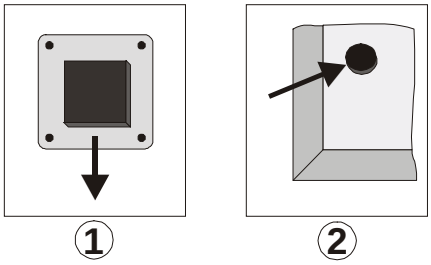


Fig. 5: Entering the System Download Mode

① Push the joystick switch down and hold it down, ② push the reset button on the top of the JControl/PLUI

(4) The mode may also be started by software calling the `run()`-Method of an instance of the built-in class `jcontrol.system.Download`.

In the first three cases, a splash screen appears as shown in *Fig. 6*. The first line of the splash screen gives information about the JControl device profile ("JControl/PLUI"). The second line shows the build date of the JCVM8, represented as format `yyyymmddhhss` (yyyy=year, mm=month, dd=day, hh=hour, ss=second). The following "+0100" in this example is optional and gives information about the time zone. The build date is also available as system property `profile.date` and used by the tools to select an appropriate device profile. The bottom line

Fig. 6: Splash screen of the system download mode

shows the parameters of the RS232 interface, fixed to 19200 bps, 8 data bits, no parity and 1 stop bit.

Using the Download Mode by Applications

The built-in class `jcontrol.system.Download` may be used to access or extend the download functionality by application software, e.g. to implement comfortable download or upload features for specific applications.

When the system download mode is started by software (see case 4), no splash screen appears and the baud rate is not fixed to 19200bps, but set to the value held by the system property `rs232.baudrate` (default: 19200bps). If data is written to the flash memory, the download mode will perform a system reset when it is quitted. Otherwise it returns to the calling application. See API documentation for more information about this class.

DISPLAY

The JControl/PLUI comes with a 128x64 pixel monochrome graphic LCD in FSTN transflective technology, backlighted by a blue LED. The display has a viewing direction of 6 o'clock and is driven by a separate display controller (Samsung S6B1713), mounted as chip-on-glass circuit on the top side of the component. To obtain a high data bandwidth, the communication between JCVM8 and display is realized by an 8 bit parallel interface.

The built-in class `jcontrol.io.Display` offers a set of methods for drawing pixels, lines, rectangles, circles, images, characters and strings on the display. It implements the interface `jcontrol.io.Graphics` for hardware abstraction. Images are supported using the pixel-based JControl Image File format (JCIF, revision 0001); fonts have to be formatted using the pixel-based JControl Font Definition format (JCFD, revision 0002). The class `jcontrol.io.Display` includes a proportional system font (8 pixel font height) by default.

For detecting the display dimensions automatically, the system property `display.dimensions` returns a string comprising the parameters `<width>x<height>x<colour_depth>`, specified by "128x64x1" for the JControl/PLUI device. The coordinates of the display are organized from left to right and from top to bottom counting from 0 to size-1, see also *Fig. 7*.

The display contrast may be adjusted by software using the system property `display.contrast`. The value is saved to Flash memory, assuring that it will be restored by the system during power-up. The blue backlight LED of the display is controlled directly by the reserved PWM channel #2 of the JCVM8. For improved hardware

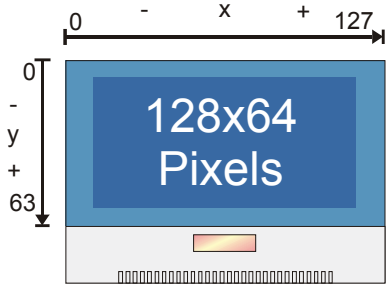


Fig. 7: Coordinates used by the LCD-class

abstraction, the external class `jcontrol.io.Backlight` is provided, enabling to set the backlight in 256 steps from 0 (off) to 255 (max brightness).

JOYSTICK SWITCH

The joystick switch enables compound actions by a single knob. It features five degrees of freedom, controlled by lever and push operations.

The built-in class `jcontrol.io.Keyboard` provides methods to read the switch on character basis, including raw access, buffered access, automatic repetition and acoustic feedback.

Fig. 8 gives an overview of the moving directions of the joystick switch. The letters 'S', 'U', 'L', 'D' and 'R' are returned by the method `read()` of the class `jcontrol.io.Keyboard` when the switch is moved to one of these positions. The letters are defined by the default keyboard map, that may be changed by an application program for software compatibility reasons.

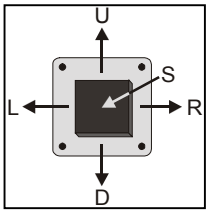


Fig. 8: Moving Directions of the Multifunction Switch

REAL TIME CLOCK (RTC)

The real time clock (RTC) of the JControl/PLUI is realized by a separate component, connected internally by the I²C bus to the JCVM8. During power-off state, the RTC is supplied by a backup gold cap, keeping the time information up-to-date.

The RTC provides year, month, day, weekday, hours, minutes and seconds. Besides the current time, an alarm time is supported by hardware. The alarm flag may be polled by software.

The built-in class `jcontrol.system.RTC` implements methods for reading and writing the current time and alarm time. A time information is represented by an instance of the built-in class `jcontrol.system.Time`, combining the fields year, month, day, weekday, hours, minutes and seconds.

The RTC is based on the Philips PCF8563 integrated circuit, clocked by a separate 32.768 kHz quartz crystal. It operates on a supply voltage down to 1.0 V and consumes approx. 250µA @V_{DD}=3.0V (T_{amb}=25°C). The RTC allocates the I²C-bus slave addresses A3h (for reading) and A2h (for writing). It may be accessed directly using the built-in class `jcontrol.comm.I2C`.

BUZZER

A piezo buzzer (mounted inside of the device) is used to generate acoustic signals. The system uses the buzzer for acoustic feedback of switch events and for signalling system exceptions. Both features may be enabled or disabled by the system properties `buzzer.systembeep` and `buzzer.keyboardbeep`. Additionally, an application software may control the buzzer using the external class `jcontrol.io.Buzzer`, implementing the interface `jcontrol.io.SoundDevice` for hardware abstraction. This class provides methods to activate the buzzer using a specified frequency (250...32767Hz) for a specified duration (in ms).

The system property `buzzer.enable` is provided to enable or disable the buzzer when it is used by an application.

Furthermore, the external class `jcontrol.toolkit.iMelody` is provided, playing complete melodies given by the iMelody-Format (`IMY`, published by the Infrared Data Association, IrDA).

The buzzer used inside the JControl/PLUI is controlled by the reserved PWM channel #3 and has a specified frequency range from 250Hz to 20kHz.

RS232 COMMUNICATION

The JControl/PLUI provides one asynchronous serial communication interface, connected by a multiplexer switch to two different communication channels. The first communication channel (named `EXTERNAL`) is used to communicate with a PC, the other (named `INTERNAL`) is used to communicate with the integrated Power-Line-BCU. The switch is controlled by the reserved

GPIO pin #8 of the JCVM8. For improved hardware abstraction, the external class `jcontrol.comm.RS232Selector` is provided to control the switch by its method `selectPort`. The parameter for this method may be either the `static integer EXTERNAL` for the Sub-D connector or `INTERNAL` for the Power-Line-BCU. *Fig. 9* shows how the two communication channels are connected to the asynchronous serial interface of the JCVM8.

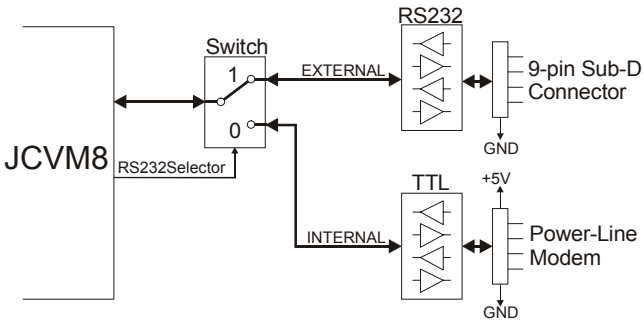
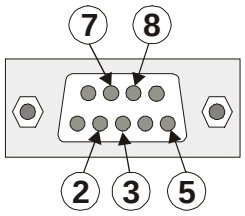


Fig. 9: Wiring of the two Communication Channels

By default (e.g. during system reset), the switch enables the communication via the RS232 interface, available as 9-pin female sub-D connector on the right edge of the device. This channel is used to upload and download application software or data from or to the flash memory, for auto identification of the JControl device and for reading or writing system properties by remote. The pin assignment of the 9-pin Sub-D connector is shown in *Fig. 10*. A simple 1:1 extension cable may be used to connect the JControl/PLUI to a PC.



Pin	Signal
2	TxD
3	RxD
5	GND
7	CTS
8	RTS

Fig. 10: Pin Assignment of the RS232 Connector (9-pin female sub-D connector)

The second communication port is wired internally and will be used for Power-Line Communication. It is connected via an opto-galvanically separated interface to the Power-Line-BCU. Refer to chapter *Power-Line Communication* (page 8 ff.) to obtain more information about this topic.

The built-in class `jcontrol.comm.RS232` provides methods for reading, writing and configuring the RS232 interface. It supports buffered read access and operates on `byte`, `char`, `string` and `utf8` basis. Automatic echoing is also supported by the `readLine()` method.

The RS232 communication interface supports 11 different baud rates, starting from 300 up to 250.000bps. The baud rate is changed using the method `setBaudrate()` of the built-in class `jcontrol.comm.RS232` (see Table 2 for a list of all valid settings). When an application attempts to set an unsupported baud rate, always the fall-back setting 19200bps is used. When no baud rate value is set by the application, the default value specified by the system property `rs232.baudrate` is used.

Baud Rate	Parameter for `setBaudrate`	Comment
300	300	
600	600	
1200	1200	
2400	2400	
4800	4800	
9600	9600	
19.200	19200	Fall-back setting
31.250	31250	MIDI
62.500	62	
125.000	125	
250.000	250	

Table 2: Supported Baud Rates

Additionally, the RS232 communication interface supports a parity bit (9th data bit) as well as flow control (by `XON/XOFF` or `RTS/CTS`). All options are defined by the current *communication*

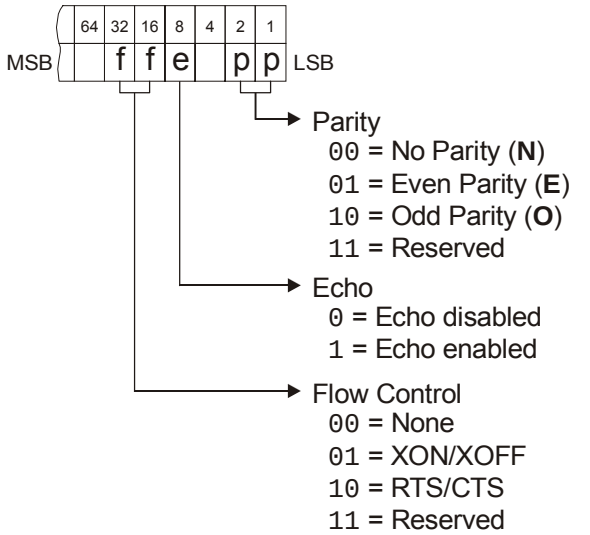


Fig. 11: RS232 Communication Parameters

parameters, configured using method `setParams()` of the built-in class `jcontrol.comm.RS232`. As shown in *Fig. 11*, the options are combined to a single bitmask. Appropriate constant field values are defined by the class `jcontrol.comm.RS232`. When the parameters are not changed by the application software, always the default settings specified by the system property `rs232.params` are used.

The following parity modes are supported: "`8N1`" (8 data bits, no parity, 1 stop bit), "`8E1`" (8 data bits, even parity, 1 stop bit) and "`8O1`" (8 data bits, odd parity, 1 stop bit). For flow control, two different modes are supported: Software flow control (by `XON/XOFF`) and hardware flow control (by `RTS/CTS`). Software flow control uses the ASCII-codes `XON` (`0x11`) and `XOFF` (`0x13`). Hardware flow control is realized by the external signals `RTS` (pin 8) and `CTS` (pin 7) of the female sub-D connector.

POWER-LINE COMMUNICATION

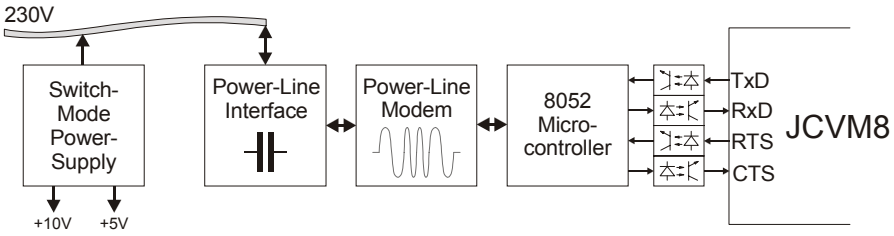


Fig. 13: Block diagram of the Power-Line-Modem

The JControl/PLUI is equipped with an integrated Konnex PL132-compliant Power-Line-BCU with a separate 8052-CPU, performing the physical layer and data link layer of the protocol stack. This reduces the software complexity and performance overhead of the JControl application software. The BCU is connected via an opto-galvanically separated 5-wire serial interface to the JCVM8 (see block diagram *Fig. 12*). Note that the multiplexer switch has to be configured to `INTERNAL` communication for establishing the link between the JCVM8 and the Power-Line-BCU (see also chapter *RS232 Communication*, page 6 ff.). The communication protocol is based on the FT1.2 specification to ensure a reliable data transfer (refer to EN60870 or IEC870). Only the signals `TxD` and `RxD` are used, `RTS` and `CTS` are reserved for future use. The communication parameters are fixed to 19200bps, 8 data bits, even parity and 1 stop bit. The external class `jcontrol.comm.FT1_2` provides a set of FT1.2-based communication methods for sending and receiving power-line messages and for configuring the BCU.

The Power-Line BCU supports six different types of messages, each preceded by a specific service code (`mcode`, see Table 3). For sending and receiving power-line messages on link layer basis, the L_DATA-frames are provided (request, indication and confirm). The remaining messages (`PC_SET_VAL` and `PC_GET_VAL`) are used to configure the BCU, e.g. to specify the number of send retries.

Name	mcode	Direction
L_DATA_req	0x11	JCVM8→BCU
L_DATA_con	0x4E	BCU→JCVM8
L_DATA_ind	0x49	BCU→JCVM8
PC_SET_VAL_req	0x46	JCVM8→BCU
PC_GET_VAL_req	0x4C	JCVM8→BCU
PC_GET_VAL_con	0x4B	BCU→JCVM8

Table 3: Messages supported by the Power-Line BCU

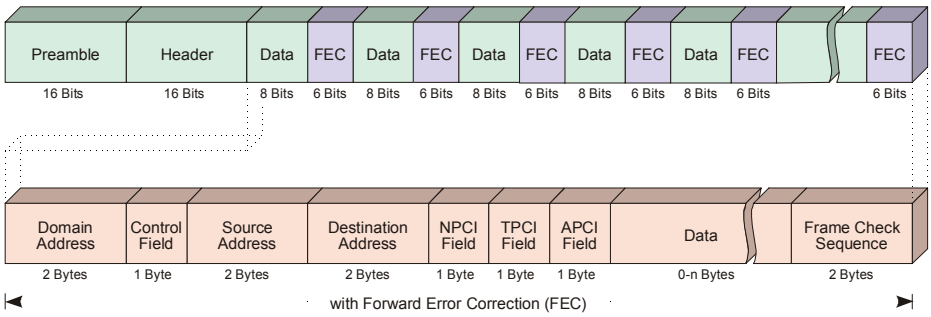


Fig. 12: Frames modulated to the Power-Line Network

In general, the L_DATA-messages exchanged between the JControl application and the BCU differ from those messages modulated to the power-line network (see also *Fig. 13*): Some data fields (preamble, header, domain address and frame check sequence) are inserted automatically by the Power-Line-BCU; therefore the JControl application has to handle the fields source address, control field, destination address, NPCI, TPCI and data only. In the following, the messages will be explained in separate subchapters.

L_DATA_req

Service	Control Field	Source Address		Destination Address		NPCI Field	TPCI Field	Data (APCI)	Data		Data
0x11	cc	0x00	0x00	high	low	nn	tt	byte 1	byte 2	.....	byte n

The message `L_DATA_req` (*req* = request) is used to send data to other power-line devices in the network. The service code is `0x11`, followed by a control field (see also *Fig. 14*). The source address (here: `0x0000`) will be replaced by the individual address of the Power-Line BCU when sending (the individual address may be configured using the control message `PC_SET_VAL.req`). The 16-bit destination address is the individual address or group address of the receiver, the fields NPCI (see also *Fig. 15*) and TPCI specify network- and transport layer information. The following data fields are used to transfer up to 15 bytes of APCI- and APDU information.

The `L_DATA_req` message is replied by a `L_DATA_con` message.

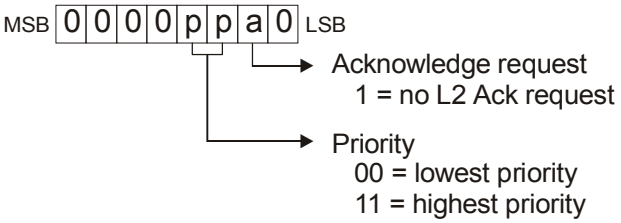


Fig. 14: Control Field (cc) of `L_Data_req` message

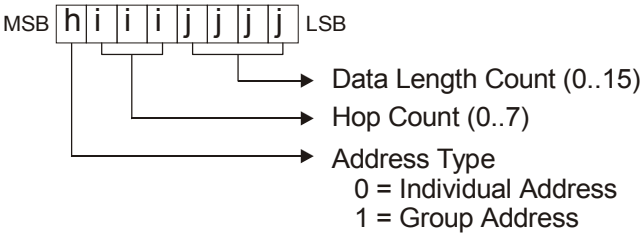


Fig. 15: NPCI (nn) Field of `L_Data_req` message

L_DATA_con

Service	Control Field	Source Address		Destination Address		NPCI Field	TPCI Field	Data (APCI)	Data (APDU)		Data (APDU)
0x4E	cc	high	low	high	low	nn	tt	byte 1	byte 2	.....	byte n

The message `L_DATA_con` (*con* = confirm) is sended by the BCU as confirmation of a `L_DATA_req` message (positive or negative confirmation). The service code is `0x4E`; most of the following fields return the same values as given by the initiating `L_DATA_req` message. The control field (*Fig. 16*) returns a confirm flag (Bit 0), signalling if the process was successful or not, and a repeat bit (Bit 5) (set to '1' if the message was repeated at least one times). Additionally, the individual address of the BCU is inserted in the source address field. The NPCI field is the same as shown for the `L_DATA_req` message (*Fig. 15*).

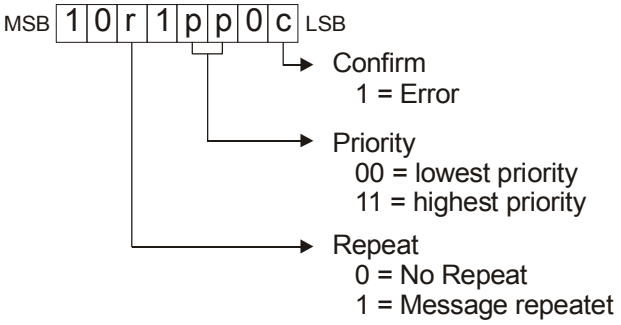


Fig. 16: Control Field (cc) of `L_Data_con` message

L_DATA_ind

Service	Control Field	Source Address		Destination Address		NPCI Field	TPCI Field	Data (APCI)	Data (APDU)		Data (APDU)
0x49	cc	high	low	high	low	nn	tt	byte 1	byte 2	.....	byte n

The `L_DATA_ind` message (*ind* = indication) is sent by the BCU when a power-line-message was received. The service code is `0x49`, followed by data fields similar to the fields described for the `L_DATA_req` message. Except the control field differs (*Fig. 17*): The repeat flag (Bit 5) indicates if the message was repeated at least one times.

Note: A message will only be received by the BCU when the address filters (Individual Address) are set properly.

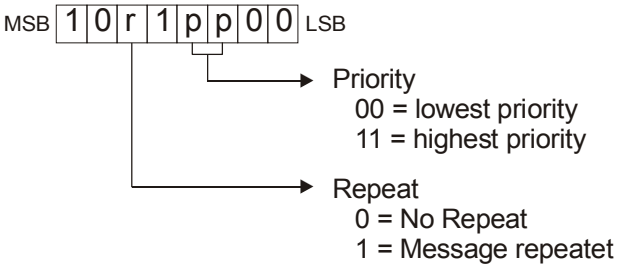


Fig. 17: NPCI (nn) Field of `L_Data_req` message

PC_SET_VAL_req

Service	Number Bytes	Memory Address		Data	Data		Data
0x46	nn	high	low	byte 1	byte 2	.....	byte n

The `PC_SET_VAL_req` message (*req* = request) is used to write various configuration parameters to the Power-Line BCU. Each parameter has the size of one byte. The service code is `0x46`, followed by the number of bytes to write and the starting memory address (`0x0000` to `0x0006`, see Table 4). The following data string will be written consecutive to the memory starting at the specified address.

Memory Address	Description	Default Value
0x0000	Domain Address Low	0x00
0x0001	Domain Address High	0x00
0x0002	Individual Address Low	0x00
0x0003	Individual Address High	0x00
0x0004	Number of Retries when NACK	0x02
0x0005	Number of Retries when BUSY	0x02
0x0006	Number of Repeats when No Acknowledge	0x02

Table 4: Memory Addresses of BCU Configuration Parameters

PC_GET_VAL_req

Service	Number Bytes	Memory Address	
0x4C	nn	high	low

The `PC_GET_VAL_req` message (*req* = request) is used to read configuration parameters from the Power-Line BCU. The service code is `0x4C`, followed by the number of bytes to read and the starting memory address. The BCU replies with a `PC_GET_VAL_ind` message.

PC_GET_VAL_ind

Service	Number Bytes	Memory Address		Data	Data		Data
0x4B	nn	high	low	byte 1	byte 2		byte n

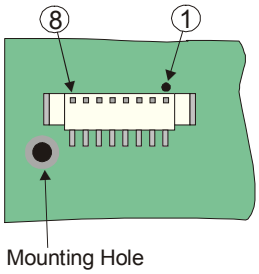
The `PC_GET_VAL_ind` message returns the configuration parameters of the Power-Line BCU, requested by a PC_GET_VAL_req message. The service code is `0x4B`, followed by the number of bytes, the starting memory address and a consecutive string of data.

EXTENSION PORT

For system extensions, an 8-pin extension port connector is available inside the device. It provides the internal operating voltage (+5V), GND and six universal I/O-pins.

Fig. 18 gives an overview of the extension port (Molex micro connector type 53398-0890). The connector is placed at the top PCB (inside the upper half of the case). For adaptation, use a Molex 51021-0800 female crimp connector.

Pin V_{cc} (1) provides the internal operating voltage of +5V in reference to GND (8). V_{cc} may be used to supply external hardware with 5V, generated by a linear regulator of JControl/PLUI. Universal I/O signals are available at pin 2 to pin 7 of the extension port.



PIN	Function
1	V_{cc} (5V)
2	GPIO #0 / PWM #0
3	GPIO #1 / PWM #1
4	GPIO #2 / ADC #0
5	GPIO #3 / ADC #1
6	GPIO #4 / ADC #2
7	GPIO #5 / ADC #3
8	GND

Connector Type:
PCB: Molex 53398-0890
Crimp: Molex 51021-0800

Fig. 18: Extension Port of the JControl/PLUI

I/O PINS (GPIO, PWM, ADC)

Six I/O pins are available for system extensions at the extension port connector, numbered from pin 2 to pin 7. Each pin may be used as digital input or output (*General Purpose I/O*, GPIO). The built-in class `jcontrol.io.GPIO` is provided to control the digital I/Os, numbered as GPIO #0 to GPIO #5. Four different configuration modes are supported:

- `FLOATING`: Standard digital input
- `PULLUP`: Digital input with integrated pull up resistor (60-240kΩ)
- `PUSHPULL`: Standard digital output
- `OPENDRAIN`: Digital output, set to high-impedance state when `HIGH`

Two of the ten GPIO pins are also connected to an integrated Pulse Width Modulator (PWM) with a resolution of up to 8 bits. This feature is controlled by the built-in class `jcontrol.io.PWM`. The generated signals are available via the PWM channels 0 and 1. The device uses a single frequency generator for all channels, hence the frequency of the channels has to be the same. Please note that every pin configured as PWM output is not available as GPIO.

The remaining four pins are connected to the internal 8-bit A/D converter and may be used as analog inputs. The built-in class `jcontrol.io.ADC` is provided to control this feature. When a pin is used as analog input, it should be configured to `FLOATING` mode using the class `jcontrol.io.GPIO`.

The following table gives an overview on the features of each pin.

Extension Port Pin	GPIO					PWM	ADC
	Channel #	Floating Input	Input With Pullup	Push Pull Output	Open Drain Output	Channel #	Channel #
2	0	✓	✓	✓	✓	0	-
3	1	✓	✓	✓	✓	1	-
4	2	✓	✓	✓	✓	-	0
5	3	✓	✓	✓	✓	-	1
6	4	✓	✓	✓	✓	-	2
7	5	✓	✓	✓	✓	-	3

Table 5: Features of pins of the extension port

I²C/SMBUS COMMUNICATION

A separate I²C communication interface is used internally to communicate with the RTC PCF8563, allocating the addresses A3h for reading and A2h for writing.

The I²C bus is a de facto standard for on-board inter integrated circuit communication. It was developed by Philips Semiconductors in the early 1980's. Many integrated circuits supporting the I²C bus. SMBus is a kind of extended I²C bus, developed by Intel in 1995 as System Management Bus. It is used e.g. in personal computers and servers for low-speed system management communications. Mostly, the SMBus is used to interconnect the sensors for temperatures, voltages, rotation speed of fans etc.

The built-in class `jcontrol.comm.I2C` provides methods for using the JControl device as bus master. It supports 7 bit and 10 bit addressing schemes as well as reading and writing single chars or byte streams. It implements a simple hardware layer, therefore any bus error and any arbitration lost results in an `IOException` after some retries. To avoid blocking, the class implements contrary to the I²C specification a bus timeout.

JCVM8 RESTRICTIONS

Not all JAVA features are implemented by the JCVM8. The following list gives an overview on the restrictions:

- Data type `int` is limited to 16 bit processing word length (not 32 bit)
- Data types `long`, `float` and `double` are not implemented. When used, one of the following two error codes is generated (context dependent):
  - `BytecodeNotSupportedError` (6)
  - `UnsupportedArrayTypeError` (9)
- The number of constants in the constant pool is limited to 255 (will be checked by the JCManager before upload)
- Cast check for primitive arrays is not supported and causes an error (`NotImplementedError`)
- It is not possible to call object methods on primitive arrays, e.g.

  ```
  new int[25].equals(myObject)
  ```

- Some exceptions can not be catched by an application, because they generate an error code. When thrown, the JCVM8 is restarted in error condition and the error handler is called (see also chapter *Error Codes*).
- Implementation of classes in the package `java.lang` is incomplete (see JControl JAVADOC)

ERROR CODES

When an exception is thrown and not handled by the application, the JCVM8 generates an error code. Some of the errors (listed in the following *Table 6*) are specific to the JCVM8 and not common in the JAVA programming language (labeled with [1]). Other error codes are *masked exceptions*, because they are generated instead of an exception (labeled with [2]).

Every error restarts the JCVM8 in error condition. Afterwards the method `onError()` of the built-in class `jcontrol.system.ErrorHandler` is invoked. More details about the error state is passed by parameters to the `onError()` method.

The built-in error handler may be overwritten by a user-defined error handler stored in Flash bank 0. See the error handler included in the SystemSetup software for demonstration.

Following table gives an overview on the error codes generated by the JCVM8.

ID	Name	Description
1	`HandleError` [1)]	Internal VM error
2	`NullPointerException` [2)]	Attempt to use `NULL` where an object is required.
3	`OutOfMemoryError`	Generated when no memory is available
4	`BytecodeNotAvailableError` [1)]	Attempt to execute an invalid bytecode

ID	Name	Description
5	BytecodeNotSupportedError [1]	Attempt to execute an unsupported bytecode, e.g. bytecodes for 64-bit arithmetic or floating point processing
6	BytecodeNotDefinedError [1]	Attempt to execute an undefined bytecode
7	ArithmeticException [2]	Exception during arithmetic processing, e.g. division by zero
8	NegativeArraySizeException [2]	Attempt to create an array with negative size
9	UnsupportedArrayTypeError [1]	Arrays of this type are not supported
10	ArrayIndexOutOfBoundsException [2]	Array index is out of bounds
11	ClassCastException [2]	Attempt to cast an object which is not of an appropriate runtime type.
12	NoCodeError [1]	Thrown when a method is called that implements no code
13	WaitForMonitorSignal [1]	Used internally by the VM
14	ExternalNativeError [1]	Generated when a native method is called that is not stored in ROM
15	FatalStackFrameOverflowError [1]	Generated when the stack size is not sufficent
16	InstantiationException [2]	Attempt to instantiate an abstract class or interface
17	IllegalMonitorStateException [2]	E.g. when a wait is called without an appropriate monitor
18	UnsatisfiedPrelinkError [1]	Error due to a failed prelinking process
19	ClassFormatError [1]	Generated when a failed while class loading
20	ClassTooBigError [1]	The size of a class exceeds the limitations
21	PreLinkError [1]	Error due to a failed prelinking process
22	PreLinkedUnresolvedError [1]	Error due to a failed prelinking process
23	UnsupportedConstantTypeError [1]	Generated when the type of a constant is not supported by the JCVM8 (`long`, `float` or `double`)
24	MalformatedDescriptorError [1]	Error while dereferencing constant pool, e.g. due to wrong class file format
25	RuntimeRefTableOverrunError [1]	More class references used than specified in a class file
26	NoSuchFieldError	Referenced field not found
27	IllegalAccessError	Tried to access a field or method from wrong scope (e.g. private)
28	NoSuchMethodError	Could not find referenced method
29	TooMuchParametersError [1]	A method uses more parameters than supported by the JCVM8 (max. 16)
30	ThrowFinalError [1]	Uncatched user defined exception. Exception name is passed to the `onError()` method.
31	NoClassDefFoundError [1]	Unable to find a class by name
32	IndexOutOfBoundsException [2]	Thrown by some methods using String or array parameters and indices that are out of bounds
33	ArrayDimensionError [1]	Generated when an array is created with more than 2 dimensions (only 1 or 2 dimensions supported)
34	DeadlockError [1]	Generated by the JCVM8 scheduler when two or more threads inheriting from each other
35	IncompatibleClassChangeError	Generated when an interface is invoked for an object, that is not implementing the interface
36	NotImplementedError [1]	Generated when an unimplemented JAVA feature is used

Table 6: Error Codes generated by the JCVM8

[1] Error codes generated exclusively by the JCVM8. Not common in the JAVA programming language.

[2] JCVM8 error codes generated by the JCVM8 instead of exceptions. Can not be handled by an exception handler. May be replaced by JAVA exceptions in future revisions of the JCVM8.

SYSTEM PROPERTIES

System properties providing specific information about the JControl device. All properties are identified by a fixed string (the content is always formatted as string). The properties may be read or written using the methods `getProperty()` and `setProperty()` of the built-in class `jcontrol.system.Management`. In download mode, the tool PropertyEdit may be used to read or write the properties by remote.

The system properties are categorized into ROM properties and non-volatile properties. ROM properties are stored in read-only memory of the device and can not be changed. Non-volatile properties are held in the upper sector of Flash bank 0 and may be changed by software.

Key	Type	Value	Description
`profile.name`	String	JControl/PLUI	JControl Profile Name
`profile.date`	String	yyyymmddhhmm	Date of JCVM build
`system.heapsize`	Int	2688	Size of internal JAVA heap memory
`flash.format`	String	“256x256x4”	Flash Organization (bytes x blocks x banks)
`io.gpiochannels`	Int	6	Number of GPIO channels
`io.pwmchannels`	Int	2	Number of PWM channels
`io.adcchannels`	Int	4	Number of ADC channels
`display.dimensions`	String	“128x64x1”	Display dimensions (width x height x colour_depth)

Table 7: ROM Properties (saved in ROM, read access only)

Key	Type	Range	Default	Description
`system.standbytimer`	Int	0..32767	60	Auto-standby time (in seconds)
`system.userbank`	Int	0..3	0	Flash bank used for user application
`rtc.poweronbank`	Int	0..3	0	Bank selected to start application after power on initiated by RTC alarm
`buzzer.enable`	Bool	true, false	true	Enable or disable buzzer to be used by application software
`buzzer.systembeep`	Bool	true, false	true	Enable or disable system sound (set independent from `buzzer.enable`)
`buzzer.keyboardbeep`	Bool	true, false	true	Enable or disable keyboard beep (set independent from `buzzer.enable`)
`display.contrast`	Int	0..255	40	LCD contrast adjustment
`rs232.params`	Int	0, 1, 8, 9	0	Bitmask holding RS232 configuration ▪ Bit 1:0 `00` = No Parity `01` = `PARITY_EVEN` enabled `10` = `PARITY_ODD` enabled ▪ Bit 3 `1` = `ECHO` enabled ▪ Bit 5:4 `00` = No flow control `01` = `FLOWCONTROL_XONOFF` enabled `10` = `FLOWCONTROL_RTSCTS` enabled
`rs232.baudrate`	Int	300, ..., 31250, 62, ..., 250	19200	Sets default RS232 Baudrate

Table 8: Non-Volatile Properties (saved in Flash, read and write access)

SUPPORTED DATA FORMATS

The device supports following data formats:

Format used for	Format suffix	Rev.	Description	Used by class	Editor
Images	JCIF	0001	JControl Image File 8-Bit pixel-based image definition format	`jcontrol.io.Display`	PictureEdit
Fonts	JCFD	0002	JControl Font Definition 8-Bit pixel-based font definition format	`jcontrol.io.Display`	FontEdit
Melodies	IMY	V1.2	iMelody Melody format specified by Infrared Data Association (IrDA)	`jcontrol.toolkit.iMelody`	MelodyEdit

Table 9: Supported Data Formats for the JControl/PLUI

The format specifications are available online at http://www.jcontrol.org.

SUPPORTED JAR LIBRARIES

The device supports base JAR Libraries as listed below:

JCVM8 Build Date	Path JControl/profiles/jar/...	Type	Description
20030109	builtin/JControl_PLUI_20030109.jar	built-in	Library of all classes provided internally
	standard/JControl_ALL_20030206_lib.jar	standard	Standard JControl API programming environment

Table 10: Base JAR libraries supported by versions of the JControl/PLUI

BUILT-IN PACKAGES

Summary of Packages

Package	Description
`jcontrol.comm`	Complex communication features for JControl.
`jcontrol.io`	Classes for basic I/O and peripheral control.
`jcontrol.lang`	Replacement classes, fundamental to the design of the JAVA programming language.
`jcontrol.system`	JControl core classes and JControl specific JAVA extensions.
`java.lang`	Provides classes that are fundamental to the design of the JAVA programming language. Subset of the standard-package `java.lang`.
`java.io`	Subset of the standard `java.io`-package (only `java.io.IOException`)

Packages in Detail

Name	Type	Description
Package jcontrol.comm		
ConsoleInputStream	Interface	Provides a set of high-level communication methods to read from a console.
ConsoleOutputStream	Interface	Provides a set of high-level communication methods to write to a console
RS232	Class	Implements RS232 communication for JControl
Package jcontrol.io		
ADC	Class	Control of JControls analog-digital converter. Used to measure the voltage at portpins connected to the internal A/D converter
BasicInputStream	Interface	Interface providing a set of low-level communication methods for reading from a stream
BasicOutputStream	Interface	Interface providing a set of low-level communication methods for writing to a stream
ComplexInputStream	Interface	Interface providing a set of high-level communication methods for reading from a stream
ComplexOutputStream	Interface	Interface providing a set of high-level communication methods for writing to a stream
Display	Class	Class to control the on-board 128x64 BW-LC-Display. Coordinates are from left to right and from top to bottom counting from 0 to size-1.
Drawable	Interface	Defines Object-behaviour for use with `jcontrol.io.Display.drawImage()`
File	Interface	Provides a set of methods for file-system access
Flash	Class	Raw access to JControl's integrated Flash memory. The methods are designed to access complete sectors of memory, not single bytes.
Graphics	Interface	Interface definition for graphics devices (e.g. offscreen images, displays, ...)
I2C	Class	Controls I^2C devices connected to JControl.
Keyboard	Class	Accesses JControl's keyboard, the joystick switch in the case of the JControl/PLUI
Portpins	Class	Controls available portpins of JControl
PWM	Class	Controls the Pulse Width Modulation outputs of JControl
Resource	Class	Implements read access to the application's resource. The resource stores additional application data like pictures, fonts, text etc.
Package jcontrol.lang		
Deadline	Class	Constructs a new JControl deadline, useful for soft real-time applications
ThreadExt	Class	Thread extensions for JControl, useful for soft real-time applications
Math	Class	Provides some simple math functions
Package jcontrol.system		
Download	Class	Manages to download new JAVA applications to a JControl module
ErrorHandler	Class	The JControl Error-Handler. May be overwritten to implement more comfortable error handlers.
Management	Class	Controls various system management functions
RTC	Class	Access to JControl's integrated Real Time Clock

Name	Type	Description
`Time`	Class	The Time object stores a date and time.
`Package java.lang`		
`Exception`	Class	The class `Exception` and its subclasses are a form of `Throwable`, indicating conditions that a reasonable application might want to catch
`Integer`	Class	The `Integer` class wraps a value of the primitive type `int` in an object. An object of type `Integer` contains a single field whose type is `int`.
`Object`	Class	Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.
`Runnable`	Interface	The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run`.
`String`	Class	The `String` class represents character strings. All string literals in JAVA programs, such as `"abc"`, are implemented as instances of this class
`Thread`	Class	A `Thread` is a thread of execution in a program. The JAVA Virtual Machine allows an application to have multiple threads of execution running concurrently.
`Throwable`	Class	The `Throwable` class is the superclass of all errors and exceptions in the JAVA language. Only objects that are instances of this class (or one of its subclasses) are thrown by the JAVA Virtual Machine or can be thrown by the JAVA `throw` statement. Similarly, only this class or one of its subclasses can be the argument type in a `catch` clause.

EXTENSION PORT SUMMARY

Pin	Name	Description
1	V_{CC}	Internal Power Supply (5V) May be used to supply external hardware [3)]
2	GPIO #0 PWM #0	▪ GPIO channel #0 ▪ Input modes: FLOATING, PULLUP ▪ Output modes: PUSHPULL, OPENDRAIN ▪ PWM channel 0
3	GPIO #1 PWM #1	▪ GPIO channel #1 ▪ Input modes: FLOATING, PULLUP ▪ Output modes: PUSHPULL, OPENDRAIN ▪ PWM channel #1
4	GPIO #2 ADC #0	▪ GPIO channel #2 ▪ Input modes: FLOATING, PULLUP ▪ Output modes: PUSHPULL, OPENDRAIN ▪ ADC channel #0
5	GPIO #3 ADC #1	▪ GPIO channel #3 ▪ Input modes: FLOATING, PULLUP ▪ Output modes: PUSHPULL, OPENDRAIN ▪ ADC channel #1
6	GPIO #4 ADC #2	▪ GPIO channel #4 ▪ Input modes: FLOATING, PULLUP ▪ Output modes: PUSHPULL, OPENDRAIN ▪ ADC channel #2
7	GPIO #5 ADC #3	▪ GPIO channel #5 ▪ Input modes: FLOATING, PULLUP ▪ Output modes: PUSHPULL, OPENDRAIN ▪ ADC channel #3
8	GND	Ground Voltage

Table 11: Extension Port Summary

[3)] Maximum supply current for external hardware: 30mA