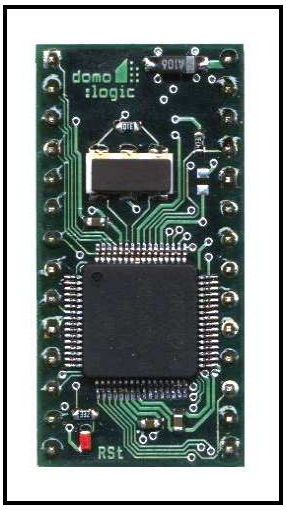
JControl/Stamp

JAVA PROGRAMMABLE CONTROL DEVICE, 64K FLASH MEMORY, ANALOG KEYBOARD SUPPORT, RS232-INTERFACE, I²C-INTERFACE, 16 GPIO AND SOFT REALTIME SUPPORT



- **Virtual Machine Core**
  - 8 Bit JAVA™ bytecode execution engine
  - 16 Bit processing word length
  - Max. 256 constant pool entries
  - 1.75kByte JAVA heap memory
  - 2 MIPS native core speed
  - Automatic garbage collection
  - Multi-threading support with extensions for soft realtime execution
- **Flash Memory**
  - up to 4 banks with 64k each
  - 128 / 256 byte sectors
  - > 10,000 erase/write cycles
- **Analog Keyboard**
  - Decoder for up to 10 keys
  - Simple and cost-effective design with resistors
  - Only one GPIO occupied
- **Power-Supply**
  - 3.3V or 5V power supply
  - Current consumption at normal operation: max. 16.5mA
- **Buzzer Support**
  - Controlled by PWM output
- **RS232**
  - 5-Wire RS232-Interface
  - 11 different baud rates from 600 up to 250.000bps
  - None, even or odd parity
  - Automatic flow control by XON/XOFF or RTS/CTS
- **I²C/SMBus Communication**
  - Master mode
  - 7 and 10 Bit addressing modes
- **I/O-Pins**
  - Up to 16 General-Purpose I/O Pins
    - 4 I/Os usable as PWM outputs
    - 8 I/Os usable as Analog inputs
- **Physical Dimensions**
  - Size: 18.0x35.8x11.8mm
  - Weight: 5g

DIMENSIONS AND CONNECTORS

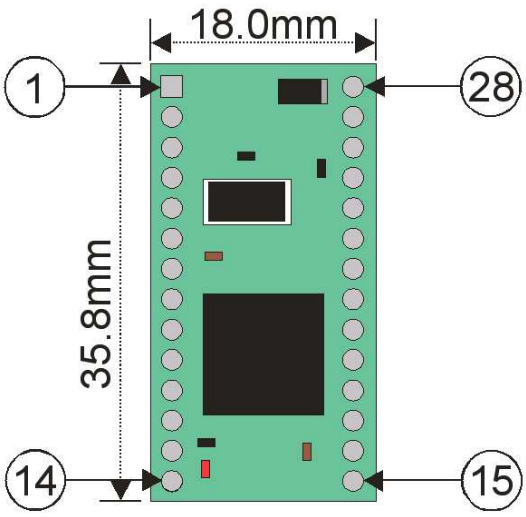


Fig. 1: Dimensions and Connectors of the JControl/Stamp

DEVICE VARIANTS

Sales Type	Power Supply	Native Core Speed	Serial Baud Rates	Flash Organization	RTC
JCSP10 S128-5-2	5V	2 MIPS	300-250.000bps	512x128x2	Soft

Table 1: Derivatives of the JControl/Stamp

GENERAL DESCRIPTION

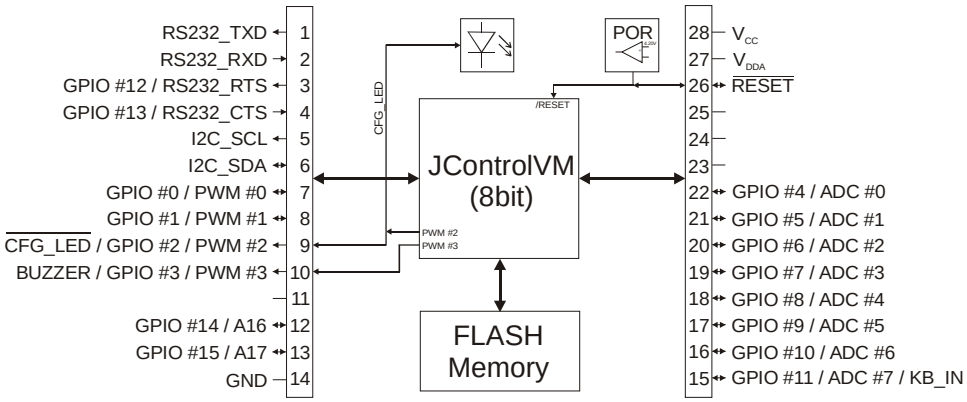


Fig. 2: JControl/Stamp Block Diagram

The JControl/Stamp is a member of the JControl device family, designed as freely programmable controlling device with analogue keyboard decoder, external buzzer control, communication ports (RS232 and I²C), general purpose I/Os, analog inputs and pulse width modulator outputs. All relevant signals are available by 28 pins at the left and right edge of the device (0.1" strip connectors). For evaluation purposes, an evaluation board is available.

The JControl/Stamp is based on the JCVM8 8 Bit JAVA™ bytecode execution engine. The core runs with 2 MIPS native speed, providing 16 Bit processing word length, 1.75kByte JAVA heap memory, automatic garbage collection and multi-threading software execution. Applications in the field of control, measurement and automation are supported by specific extensions for soft-realtime processing.

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 analog keyboard, Flash memory etc. 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, which is organized as one to four banks of 64kByte each. The banks may be used to store application software or non-volatile data.

Various informations about the specific JControl device and its current state is available by accessing the *system properties*. In download mode, the 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/SmartDisplay is powered by 5V DC or 3.3V DC, connected to the pins 14 (GND) and 28 (V_{CC}) of the device. To ensure a reliable start up phase, an integrated power-on reset generator (POR) holds the reset signal of the JCVM8 while the supply voltage is below the threshold voltage of 4.50V or 3.00V resp. When the supply voltage exceeds the threshold voltage, the reset signal is released and the initialization sequence of the JCVM8 is executed. When finished, the JAVA-application stored in Flash bank 0 is started.

FLASH MEMORY ORGANIZATION

Depending on the device variant, the JControl/Stamp offers one to four banks of 64k Flash memory for application software or non-volatile data, labeled as Flash bank 0 to Flash bank 3. For devices with up to two flash banks, the memory is organized as 512 sectors by 128 bytes, numbered from sector 0 to sector 511. For devices with 4 Flash banks, the memory is organized as 256 sectors by 256 bytes. The Flash memory's organization may be detected automatically by reading the system property `flash.format`. The returned string comprises of

the parameters <number of sectors>x<bytes per sector>x<number of banks> (e.g. "512x128x1" for the JControl/Stamp device with one flash bank).

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.

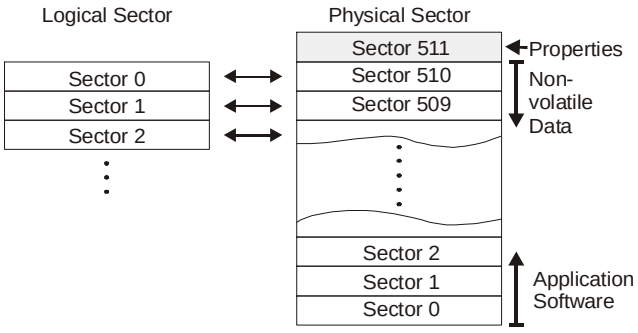


Fig. 3: Internal Structure of Flash bank 0
(For Devices with 1 or 2 Flash Banks with 512 Sectors per Bank)

Fig. 3 gives an overview of bank 0's internal structure: Application software is written upwards, starting at physical sector 0 and non-volatile data is stored downwards, starting at physical sector 510 (respective 254 for devices with four flash banks). This procedure reduces the possibility of resource conflicts between application software 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 510 of the Flash memory, access to logical sector 1 to the physical sector 509 and so on. The uppermost sector of bank 0 (sector 511) is used to hold non-volatile system properties. The same principle is also used for Flash Bank 1, except that the uppermost sector is not holding the system properties.

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 application software to and downloading data from the Flash memory by a host computer, 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) Analog Keyboard.

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

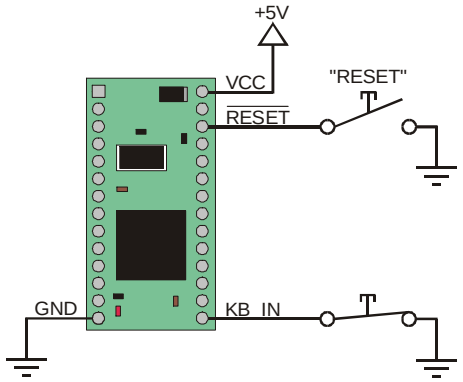


Fig. 4: Entering the System Download Mode

① Pull signal KB_IN to GND (Pin 15), ② activate RESET signal for more than 10ms (Pin 26)

The system download mode may be enforced by pulling Pin 15 (GPIO #11/KB_IN) to GND while resetting the device (as shown in *Fig. 4*). Refer to the chapter covering the

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), the baud rate is set to the value held by the system property `rs232.baudrate` (default: 19200bps). When quitting, the download mode performs a system reset when data was written to the flash memory. Otherwise it returns to the calling application. See API documentation for more information about this class.

ANALOG KEYBOARD

The JControl/Stamp provides a decoder for *analog keyboards* with up to 10 keys. Analog keyboards are designed as switched resistor ladders, generating a specific voltage for each key. This mechanism reduces the complexity required to realize a keyboard to a minimum.

For the JControl/Stamp, GPIO #11 is used for connecting the analog keyboard (this pin is also labeled as `KB_IN`). GPIO #11 is internally connected to the ADC #7 pin of the JControl/GUI-engine. The system software measures the voltage at ADC #7 every 16ms, corresponding to a keyboard request rate of 62.5Hz. The built-in class `jcontrol.io.Keyboard` provides methods for reading the switch on character basis, including raw access, buffered access, automatic repetition and acoustic feedback.

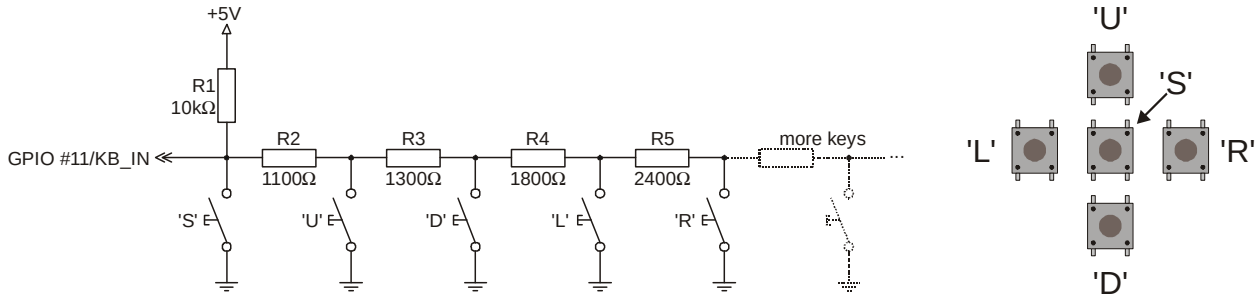


Fig. 5: Schematic of a cursor-control analog key panel

Fig. 5 shows the schematic of a simple cursor-control key panel connected to the JControl/Stamp, realizing the keys 'up', 'down', 'left', 'right' and 'select'. The pull-up resistor R1 (10kΩ) is used to apply a quiescent voltage of V_{DDA} (analog reference voltage) to the analog channel, representing the passive state when all keys are released. Each keypress creates a specific voltage divider, composed by R1 and a chain of resistors from R2 to the resistor connected to the corresponding key. The resulting voltage is measured via ADC #7 (KB_IN).

Table 2 lists the resistor values for an analog keyboard with up to 10 keys, using a pull-up resistor R1 of 10kΩ. The voltage created by the voltage dividers is increased by steps of $V_{DDA}/10$, starting at 0V. Because of resistor tolerances, the resulting voltages and hence the measured ADC value may differ in real applications. Hence, the integrated keyboard decoder uses thresholds between two theoretical values for key detection.

The keys decoded by the key panel shown in *Fig. 5* are 'up', 'down', 'left', 'right' and 'select', corresponding to the letters 'U', 'D', 'L', 'R' and 'S'. The letters are returned by the method `read()` of the class `jcontrol.io.Keyboard`, when one of the keys is pressed. The letters are defined by the default keyboard map, that may be changed by an application program for software compatibility reasons.

Note that the keys are prioritized, i. e. always the key with the lowest order number is decoded, if various keys are pressed simultaneously. The first key (letter 'S') is also used to enter the download mode when pressed during reset.

Key Nr.	Letter	R	Resistor Value	ADC Value
1	'S'		0Ω	0
2	'U'	R2	1100Ω	25
3	'D'	R3	1300Ω	50
4	'L'	R4	1800Ω	76
5	'R'	R5	2400Ω	102
6	'N'	R6	3300Ω	127
7	'P'	R7	5100Ω	153
8	'E'	R8	8200Ω	179
9	'H'	R9	16000Ω	204
10	'X'	R10	51000Ω	230

Table 2: Resistor values for the Analog Keyboard

Analog keyboards are not suitable for silicone rubber keys, because of their varying and pressure-dependent contact resistances. Use external hardware to reduce the contact resistance in this case, e.g. logic buffers.

REAL TIME CLOCK (RTC)

The JControl/Stamp implements a software emulated Real Time Clock (RTC), controlled by the system software. Hence, this "RTC" is clocked by the on-board ceramic resonator. It provides year, month, day, weekday, hours, minutes and seconds. Besides the current time, an alarm time is also provided. When the current time reaches the alarm time, a dedicated alarm flag is set.

The built-in class `jcontrol.system.RTC` implements methods for reading and writing the current time and the 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.

BUZZER CONTROL

The JControl/Stamp supports an external buzzer, connectable to pin 10 of the device. The buzzer may be used for acoustic signals generated by the system or by an application.

The system will use the buzzer for acoustic feedback on 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 output is connected internally to PWM channel #3.

As a buzzer, a simple piezo element may be used, connected directly to Pin 10. If a magnetic loudspeaker is used, please provide a transistor for boosting the output signal and a free wheeling diode to block reverse voltages generated by the coil.

RS232 COMMUNICATION

The JControl/Stamp provides a serial communication interface with CMOS/TTL levels. The signals are available at pin 1 (output signal `TXD`) and pin 2 (input signal `RXD`) of the device. Optionally, two signals for flow control are available at pin 3 (output signal `RTS`) and pin 4 (input signal `CTS`).

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, including the MIDI-baud rate of 31250bps. The baud rate is changed using the method `setBaudrate()` of the built-in class `jcontrol.comm.RS232` (see Table 3 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.

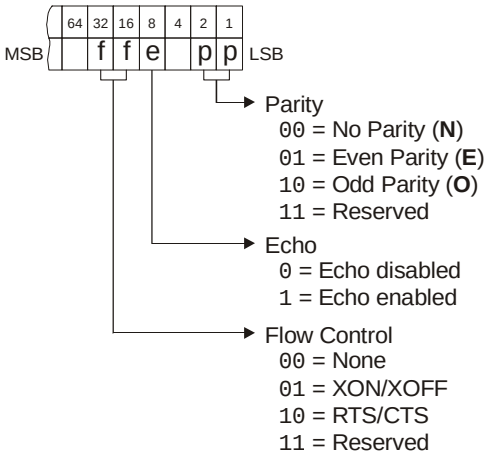
Baud Rate	Parameter for `setBaudrate`	Comment
600	600	
1200	1200	
2400	2400	
4800	4800	
9600	9600	
19.200	19200	Fall-back setting
31.250	31250	MIDI
38.400	38	
62.500	62	
125.000	125	
250.000	250	

Table 3: 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 parameters*, configured using method `setParams()` of the built-in class

`jcontrol.comm.RS232`. As shown in *Fig. 6*, 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 `RS232_RTS` (pin 3) and `RS232_CTS` (pin 4) of the JControl/Stamp.



I/O PINS (GPIO, PWM, ADC)

The JControl/Stamp provides 16 universal I/O signals for external hardware control. Each signal is available as General Purpose I/O (GPIO), numbered as GPIO #0 to GPIO #15. The built-in class `jcontrol.io.GPIO` is provided to control the I/Os, supporting four different configuration modes:

- `FLOATING`: Standard digital input
- `PULLUP`: Digital input with integrated pull up resistor (80kΩ-240kΩ ; typically 120kΩ ; can not be influenced)
- `PUSHPULL`: Standard digital output
- `OPENDRAIN`: Digital output, set to high-impedance state when `HIGH`

The output current for any pin must not exceed 25mA, indepently of its usage (either source or sink).

Four of the 16 GPIO pins are also connected to an integrated Pulse Width Modulator (PWM) which provides 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 to 3. The device uses a single frequency generator for all channels, hence the frequency of the channels has to be the same, but the duty cycle of each PWM channel may be adjusted individually. Please note that each pin configured as PWM output is not available as GPIO.

NOTE: PWM channel 2 is hardwired to the on-board LED, which may affect peripheral hardware connected to this pin.

Furthermore, eight 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 reference voltage for the ADC channels must be connected to pins V_{DDA} (high potential; pin 27) and GND (low potential; pin 14) and may not exceed the supply voltage.

Table 4 gives an overview on the features of each pin. Two of to the listed GPIOs (#12 and #13) are provided to control the RS232 hardware flow signals `RTS` and `CTS`. Refer to chapter RS232 Communication for more information about this topic.

Availability of GPIO #14 and GPIO #15 depends on the format of the equipped flash memory.

Device Pin	GPIO #	PWM #	ADC #	Alternate function	GPIO configurations [1)]
7	0	0	-	-	FI, PU, PP, OD
8	1	1	-	-	FI, PU, PP, OD
9	2	2	-	/CFG_LED	FI, PU, PP, OD
10	3	3	-	BUZZER	FI, PP, OD
22	4	-	0	-	FI, PU, PP, OD
21	5	-	1	-	FI, PU, PP, OD
20	6	-	2	-	FI, PU, PP, OD

Device Pin	GPIO #	PWM #	ADC #	Alternate function	GPIO configurations [1]
19	7	-	3	-	FI, PU, PP, OD
18	8	-	4	-	FI, PU, PP, OD
17	9	-	5	-	FI, PU, PP, OD
16	10	-	6	-	FI, PU, PP, OD
15	11	-	7	KB_IN	FI, PU, PP, OD
3	12	-	-	RS232_RTS	PU
4	13	-	-	RS232_CTS	FI, PU, PP, OD
12	14[2]	-	-	A16	FI, PU, PP, OD
13	15[3]	-	-	A17	FI, PP, OD

Table 4: Features of universal I/O pins

[1] FI = **FLOATING** input ; PU = Input with internal **PULLUP** resistor ; PP = **PUSHPULL** output ; OD = **OPENDRAIN** output
[2] **GPIO #14** only available in combination with flash format **512x128x1**
[3] **GPIO #15** only available in combination with flash format **512x128x1** or **512x128x2**

I²C COMMUNICATION

A I²C/SMBus communication interface is available at the port of the JControl/Stamp.

The I²C bus is a de facto standard for on-board inter-IC communication. It was developed by Philips Semiconductors in the early 1980's. Many integrated circuits are 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 a few retries. To avoid blocking, the class implements a bus timeout (in contrast to the I²C bus specification).

The signal `I²C_SCL` (pin 5) is the clock signal of the I²C bus (or `SMBCLK` of SMBus). The signal `I²C_SDA` (pin 6) is the data signal of the I²C bus (or `SMBDAT` of SMBus).

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 5*) 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
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 by an invalid class
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)

ID	Name	Description
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 and 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 5: 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/Stamp"	JControl Profile Name
`profile.date`	String	"{yyyyMMddhhmm}"	Date of JCVM build
`system.heapsize`	Int	1792	Size of internal JAVA heap memory
`flash.format`	String	"512x128x1"	Flash Organization (bytes x blocks x banks)
`io.gpiochannels`	Int	16	Number of GPIO channels
`io.pwmchannels`	Int	4	Number of PWM channels
`io.adcchannels`	Int	8	Number of ADC channels

Table 6: ROM Properties (saved in ROM, read access only)

Key	Type	Range	Default	Description
`system.userbank`	Int	0..1	0	Flash bank used for user application
`rtc.poweronbank`	Int	0..1	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`)
`rs232.params`	Int	[1]	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	[2]	19200	Sets default RS232 Baudrate

Table 7: Non-Volatile Properties (saved in Flash, read and write access)

[1] see: Figure 6 (RS232 Communication)

[2] see: Table 3 (RS232 Communication)

SUPPORTED DATA FORMATS

The device supports following data formats:

Format used for	Format suffix	Rev.	Description	Used by class	Editor
Melodies	IMY	V1.2	iMelody Melody format specified by Infrared Data Association (IrDA)	`jcontrol.toolkit.iMelody`	MelodyEdit

Table 8: Supported Data Formats for the JControl/Stamp

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

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
`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.
I2C	Class	Controls I^2C devices connected to JControl.
Keyboard	Class	Accesses JControl's keyboard, the analog keyboard in the case of the JControl/Stamp
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
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.

MECHANICAL DATA

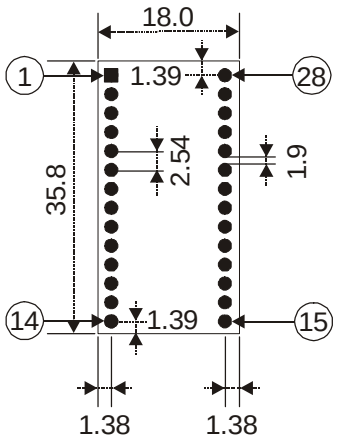


Fig. 7: Mechanical Data of JControl/Stamp
(All sizes in mm)

PIN ASSIGNMENT

Pin	Name	Description
1	RS232_TXD	Transmit Data output of RS232 interface
2	RS232_RXD	Receive Data input of RS232 interface

3	GPIO #12
	RS232_RTS
		▪ GPIO channel #12
		  ▪ Input modes: PULLUP
		  ▪ Output modes: -
		▪ Ready To Send handshake output of RS232 interface

4	GPIO #13
	RS232_CTS
		▪ GPIO channel #13
		  ▪ Input modes: FLOATING or PULLUP
		  ▪ Output modes: PUSHPULL or OPENDRAIN
		▪ Clear To Send handshake input of RS232 interface

5	I²C_SCL	Clock Signal of I²C-Bus (SMBCLK of SMBus)
6	I²C_SDA	Data Signal of I²C-Bus (SMBDAT of SMBus)

7	GPIO #0
	PWM #0
		▪ GPIO channel #0
		  ▪ Input modes: FLOATING or PULLUP
		  ▪ Output modes: PUSHPULL or OPENDRAIN
		▪ PWM channel #0

8	GPIO #1
	PWM #1
		▪ GPIO channel #1
		  ▪ Input modes: FLOATING or PULLUP
		  ▪ Output modes: PUSHPULL or OPENDRAIN
		▪ PWM channel #1

9	GPIO #2
	PWM #2
	/CFG_LED
		▪ GPIO channel #2
		  ▪ Input modes: FLOATING or PULLUP
		  ▪ Output modes: PUSHPULL or OPENDRAIN
		▪ PWM channel #2
		▪ CONFIG LED control output
		  ▪ Internally connected to PWM channel #2 and to config LED

10	GPIO #3
	PWM #3
	BUZZER
		▪ GPIO channel #3
		  ▪ Input modes: FLOATING
		  ▪ Output modes: PUSHPULL or OPENDRAIN
		▪ PWM channel #3
		▪ Buzzer control output
		  ▪ Internally connected to PWM channel #3

Pin	Name	Description
11	n/c	Reserved for future use
12	GPIO #14 A16	▪ GPIO channel #14 **(only available for flash format 512x128x1)** ▪ Input modes: FLOATING or PULLUP ▪ Output modes: PUSHPULL or OPENDRAIN ▪ Address signal 16
13	GPIO #15 A17	▪ GPIO channel #15 **(only available for flash format 512x128x1 or 512x128x2)** ▪ Input modes: FLOATING ▪ Output modes: PUSHPULL or OPENDRAIN ▪ Address signal 17
14	GND	Ground Voltage (also low potential of the analog reference voltage)
15	GPIO #11 ADC #7 KB_IN	▪ GPIO channel #11 ▪ Input modes: FLOATING or PULLUP ▪ Output modes: PUSHPULL or OPENDRAIN ▪ ADC channel #7 ▪ Analog Keyboard Input
16	GPIO #10 ADC #6	▪ GPIO channel #10 ▪ Input modes: FLOATING or PULLUP ▪ Output modes: PUSHPULL or OPENDRAIN ▪ ADC channel #6
17	GPIO #9 ADC #5	▪ GPIO channel #9 ▪ Input modes: FLOATING or PULLUP ▪ Output modes: PUSHPULL or OPENDRAIN ▪ ADC channel #5
18	GPIO #8 ADC #4	▪ GPIO channel #8 ▪ Input modes: FLOATING or PULLUP ▪ Output modes: PUSHPULL or OPENDRAIN ▪ ADC channel #4
19	GPIO #7 ADC #3	▪ GPIO channel #7 ▪ Input modes: FLOATING or PULLUP ▪ Output modes: PUSHPULL or OPENDRAIN ▪ ADC channel #3
20	GPIO #6 ADC #2	▪ GPIO channel #6 ▪ Input modes: FLOATING or PULLUP ▪ Output modes: PUSHPULL or OPENDRAIN ▪ ADC channel #2
21	GPIO #5 ADC #1	▪ GPIO channel #5 ▪ Input modes: FLOATING or PULLUP ▪ Output modes: PUSHPULL or OPENDRAIN ▪ ADC channel #1
22	GPIO #4 ADC #0	▪ GPIO channel #4 ▪ Input modes: FLOATING or PULLUP ▪ Output modes: PUSHPULL or OPENDRAIN ▪ ADC channel #0
23	n/c	Reserved for future use
24	n/c	Reserved for future use
25	N.C. n/c	Reserved for future use
26	/RESET	Reset input, active low
27	V_{DDA}	Reference voltage for ADC channels (high potential)
28	V_{CC}	Power Supply (5V or 3.3V DC)

Table 9: Pin Assignment of JControl/Stamp

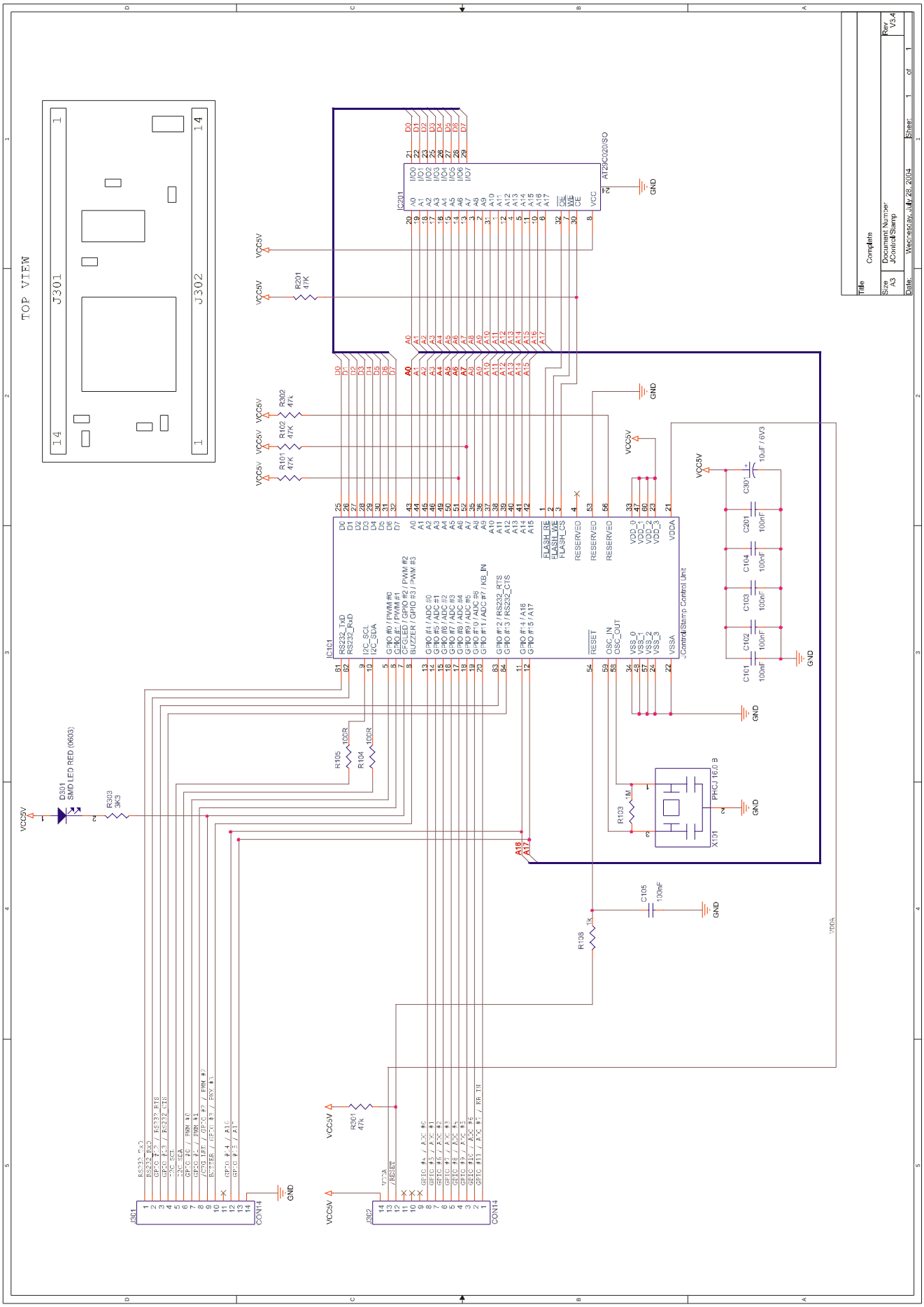


Fig. 8: Schematic of the JControl/Stamp

NOTES