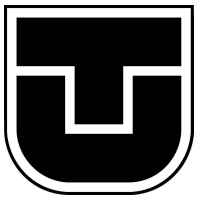

TECHNICKÁ UNIVERZITA V KOŠICIACH

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

emuStudio

Peter Jakubčo

Príloha B
Diplomová práca
Košice, 2009

Systémová príručka

Thesis supervisor and consultant: *ing. Slavomír Šimoňák, PhD.*

Contents

Package plugins

Package Contents *Page*

1.1 Interfaces

1.1.1 INTERFACE **IPlugin**

Root interface for all plugins, defines description, version, name and copyright information.

1.1.1.1 DECLARATION

public interface IPlugin

1.1.1.2 METHODS

- *destroy*
  `public void` **destroy** `( )`
  - **Usage**
    - This method is called immediately after user closes the emulator. It means, that after return from this method instance of the plugin will be destroyed. It should contain some clean-up or destroy code for GUIs, stop timers, threads, etc.

- *getCopyright*
  `public String` **getCopyright** `( )`
  - **Usage**
    - Get legal copyright of the plugin.
  - **Returns** - legal copyright

- *getDescription*
  `public String` **getDescription** `( )`
  - **Usage**
    - Get short description of the plugin. It should not be used as a manual :-)
  - **Returns** - short description of the plugin

- *getHash*
  `public long` **getHash** `( )`
  - **Usage**
    - The ”hash” will be assigned to a plugin in their run-time, at once by its constructor. The plugins will identify themselves using given hash.

  - **Returns** - hash that was assigned to the plugin

- *getTitle*
  `public String` **getTitle**`( )`
  - **Usage**
    - Get name of the plugin. This name is only ”marketing”-name, it has no relevance for identifying the plugin. This name is shown in ”Preview Configuration” in main module, and if the plugin is device, also in ”devices” section in panel ”emulator” in the main module.
  - **Returns** - name of the plugin

- *getVersion*
  `public String` **getVersion**`( )`
  - **Usage**
    - Get version of the plugin. String is returned, so version can be in arbitrary format. This version should not to be used for identifying the plugin. Better for this purpose is checking plugin’s context version and ID.
  - **Returns** - version of the plugin

- *reset*
  `public void` **reset**`( )`
  - **Usage**
    - Perform a reset of this plugin. Reset process depends on the type of the plugin.

- *showSettings*
  `public void` **showSettings**`( )`
  - **Usage**
    - Every plugin should have its own GUI for settings manipulation. This method invokes it.

1.1.2 INTERFACE **IContext**

Parent interface for all contexts.

1.1.2.1 DECLARATION

public interface IContext

1.1.2.2 METHODS

- *getHash*
  `public String` **getHash** `( )`
  - **Usage**
    - Hash string (doesn't matter how long) is computed from names of all methods. Hash is computed from a string that consists of alphabetically sorted names of all context methods with their return type and parameter types.
      A single method is written in the following way: "type name(parameter_type1,parameter_type2,...)"
      Methods are then sorted alphabetically in ascending, and joined together with separator ";", like this: "method1;method2;..."
      For example: "String getID();void setRAM(int,int)"
      Then the string has to be hashed by md5 hash.
      The hash is used for identifying the context by other plugins.
  - **Returns** - hash of all context methods

- *getID*
  `public String` **getID** `( )`
  - **Usage**
    - Return unique ID of this context. This can be any string identifying concrete context. Usually it is related with kind of context (e.g. "cpu8080","mitsSIO-2",...) . Other plugins can identify the context by recognizing of its ID.
  - **Returns** - ID of this context.

1.1.3 INTERFACE **ISettingsHandler**

Interface for all plugins, it perform methods for reading/storing settings. It is implemented by the main module and plugins obtain its object by an parameter in initialization process.

1.1.3.1 DECLARATION

```
public interface ISettingsHandler
```

1.1.3.2 METHODS

- *readSetting*
  `public String` **readSetting**`( long` **pluginHash**`, java.lang.String` **settingName** `)`
  - **Usage**
    - Read specified setting from configuration file. Setting can be arbitrary. It uses configuration file that user chosen after start of the emulator.
  - **Parameters**
    - `pluginHash` - hash of a plugin
    - `settingName` - name of wanted setting (without spaces)
  - **Returns** - setting if it exists (as a `String`), or `null` if not

---

- *removeSetting*
  `public void` **removeSetting**`( long` **pluginHash**`, java.lang.String` **settingName** `)`
  - **Usage**
    - Remove specified setting to from a configuration file. Be careful, setting can be arbitrary. It uses configuration file that user chosen after start of the emulator.
  - **Parameters**
    - `pluginHash` - hash of a plugin
    - `settingName` - name of wanted setting (without spaces) to be removed

---

- *writeSetting*
  `public void` **writeSetting**`( long` **pluginHash**`, java.lang.String` **settingName**`, java.lang.String` **val** `)`
  - **Usage**

    * Write specified setting to a configuration file. Setting can be arbitrary. It uses configuration file that user has chosen after start of the emulator.

– **Parameters**

    * `pluginHash` - hash of a plugin
    * `settingName` - name of wanted setting (without spaces) to be written
    * `val` - value of the setting (has to be `String` type)

Package plugins.memory

Package Contents *Page*

2.1 Interfaces

2.1.1 INTERFACE **IMemory**

This is the main interface that memory plugin should implement.

2.1.1.1 DECLARATION

public interface IMemory
implements plugins.IPlugin

2.1.1.2 METHODS

- *getContext*
  `public IMemoryContext` **getContext**`( )`
  - **Usage**
    - Gets memory context. Via memory context devices and CPU performs access to memory cells. If memory supports some special techniques (e.g. banking, segmentation, paging, etc.), the context should be extended by new one, that's interface will be public to all interested plugnis or CPUs.
  - **Returns** - memory context object

- *getProgramStart*
  `public int` **getProgramStart**`( )`
  - **Usage**
    - Gets program's start address. The start address is set invoking memory's method `IMemory.setProgramStart()` by main module when compiler finishes compilation process of a program and if the compiler know the starting address. This address is used by main module for CPU reset process.
  - **Returns** - program's start address in memory

- *getSize*
  `public int` **getSize**`( )`
  - **Usage**
    - Gets size of memory. If memory uses some techniques as banking, real size of the memory is not computed. It is only returned a value set in architecture configuration.

  - **Returns** - basic size of the memory

- *initialize*
  `public boolean` **initialize**`( int` **size,** `plugins.ISettingsHandler` **sHandler** `)`
  - **Usage**
    - Perform initialization process of memory. The memory should physically create the memory - e.g. as an array or something similar. Memory can't use CPU nor devices. It is accessed BY them.
  - **Parameters**
    - `size` - size of the memory, set in architecture configuration
    - `sHandler` - settings handler object. Memory can use this for accessing/storing/removing its settings.
  - **Returns** - true if initialization process was successful, false otherwise

---

- *setProgramStart*
  `public void` **setProgramStart**`( int` **address** `)`
  - **Usage**
    - Sets program start address. This method is called by main module when compiler finishes compilation process and return known start address of compiled program. This start address is then used by CPU, in reset operation - PC (program counter, or something similar) should be set to this address, accessible via `IMemoryContext.getProgramStart()` method.
  - **Parameters**
    - `address` - starting memory position (address) of a program

- *showGUI*
  `public void` **showGUI**`( )`
  - **Usage**
    - Show GUI of a memory. Every memory plugin should have a GUI, but it is not a duty.

2.1.2 INTERFACE **IMemoryContext**

Interface provides a context for operating memory. It supports basic methods, but if memory wants to support more functionality, this interface should be extended by plugin programmer and he should then make it public, in order to plugins have access to it.

The context is given to plugins (compiler, CPU, devices), that are connected to the memory and they communicate by invoking following methods.

2.1.2.1 DECLARATION

public interface IMemoryContext
implements plugins.IContext

2.1.2.2 METHODS

- *addMemoryListener*
  `public void` **addMemoryListener**`(`
  `plugins.memory.IMemoryContext.IMemListener` **listener** `)`
  - **Usage**
    - Adds the specified memory listener to receive memory events from this memory. Memory events occur even if single cell is changed in memory. If listener is `null`, no exception is thrown and no action is performed.
  - **Parameters**
    - `listener` - the memory listener

- *clearMemory*
  `public void` **clearMemory**`( )`
  - **Usage**
    - Clears the memory.

- *getDataType*
  `public Class` **getDataType**`( )`
  - **Usage**
    - Get the type of transferred data. As you can see, methods `read` and `write` use `Object` as the data type. This method should make the data type specific.
  - **Returns** - data type of transferred data

- *read*
  `public Object` **read**`( int` **from** `)`
  - **Usage**
    - Read one cell from a memory.
  - **Parameters**
    - `from` - memory position (address) of the read cell
  - **Returns** - read cell (don't have to be a byte, therefore its just `Object`)

- *readWord*
  `public Object` **readWord**`( int` **from** `)`
  - **Usage**
    - Read two cells from a memory at once. Implementation of return value is up to plugin programmer (concatenation of the cells). If cells in memory are pure bytes (java type is e.g. `short`), concatenation can be realized as (in small endian): `result = (mem[from]&0xFF) | ((mem[from+1]<<8)&0xFF);` and in big endian as: `result = ((mem[from]<<8)&0xFF) | (mem[from+1]&0xFF);`
  - **Parameters**
    - `from` - memory position (address) of the read cells
  - **Returns** - two read cells

- *removeMemoryListener*
  `public void` **removeMemoryListener**`(`
  `plugins.memory.IMemoryContext.IMemListener` **listener** `)`
  - **Usage**
    - Removes the specified memory listener so that it no longer receives memory events from this memory. Memory events occur even if single cell is changed in memory. If listener is `null`, no exception is thrown and no action is performed.
  - **Parameters**
    - `listener` - the memory listener to be removed

- *write*
  `public void` **write**`( int` **to,** `java.lang.Object` **val** `)`
  - **Usage**
    - Write one cell-size (e.g. byte) data to a cell to a memory on specified location.
  - **Parameters**
    - `to` - memory position (address) of the cell where data will be written
    - `val` - data to be written

- *writeWord*
  `public void` **writeWord**`( int` **to,** `java.lang.Object` **val** `)`

- **Usage**
  - Write two cell-size (e.g. word or two bytes) data to a cell to a memory on specified location. Implementation of data value is up to plugin programmer (concatenation of the cells) and have to be understandable by memory.
- **Parameters**
  - `to` - memory position (address) of the read cells
  - `val` - two cells in one `Object` value

2.1.3 INTERFACE **IMemoryContext.IMemListener**

The listener interface for receiving memory events. The class that is interested in processing a memory event implements this interface, and the object created with that class is registered with a memory, using the memory's `addMemoryListener` method. Memory events occur even if single cell is changed in memory and then is invoked `memChange` method.

2.1.3.1 DECLARATION

public static interface IMemoryContext.IMemListener **implements** java.util.EventListener

2.1.3.2 METHODS

- *memChange*
  `public void` **memChange**`( java.util.EventObject` **evt**`, int` **adr** `)`
  - **Usage**
    - This method is invoked when memory event is occurred - when a single cell is changed.
  - **Parameters**
    - `evt` - event object
    - `adr` - memory position (address) of changed cell

Package plugins.cpu

Package Contents *Page*

Interfaces

3.1 Interfaces

3.1.1 INTERFACE **ICPU**

Interface that covers CPU. This is the main interface that CPU plugin has to implement.

3.1.1.1 DECLARATION

public interface ICPU
implements plugins.IPlugin

3.1.1.2 FIELDS

- public static final int STATE_STOPPED_NORMAL
  - CPU is stopped (naturally or by user) and should not be run until its reset.
- public static final int STATE_STOPPED_BREAK
  - CPU is in breakpoint state (paused).
- public static final int STATE_STOPPED_ADDR_FALLOUT
  - CPU is stopped because of address fallout error. It should not be run until its reset.
- public static final int STATE_STOPPED_BAD_INSTR
  - CPU is stopped because of instruction fallout (unknown instruction) error. It should not be run until its reset.
- public static final int STATE_RUNNING
  - CPU is running.

3.1.1.3 METHODS

- *execute*
  `public void` **execute**`( )`
  - **Usage**

    * Runs CPU emulation. Change state of CPU to ”running” and start instruction fetch/decode/execute loop. Best for this purpose is to create a separate thread that runs permanently and protect emulator from ”freeze”, the cause of started instruction execution loop. It should be possible to control the thread in future by other methods: `pause()` and `stop()`. Therefore after CPU's run the execution should return from this method. It can be assumed that while CPU is in run state, main module won't allow to call method `step()`.
      In CPU run state (a good CPU) performs right timing for instructions. Debug window isn't updated after execution of each instruction, so execution loop should be faster as it is by calling `step` method.

- *getBreakpoint*
  `public boolean` **getBreakpoint**`( int` **pos** `)`
  - **Usage**
    * Determine breakpoint on specified address. It should be called only if breakpoints are supported (`isBreakpointSupported()`), otherwise the method should return false always.
  - **Parameters**
    * `pos` - memory position (address), from where we try to determine breakpoint
  - **Returns** - true if breakpoint exists on specified address, false otherwise

---

- *getContext*
  `public ICPUContext` **getContext**`( )`
  - **Usage**
    * Get CPU context. CPU context is an object that implements basic `ICPUContext` interface. Often this interface is extended by another (not supported by this library), a concrete context for concrete CPU and gives more functionality than basic one. Context is a place, where plugin programmer should implement unsupported, but needed methods and then he should make public its interface. Plugins connected to CPU get its context as a parameter in plugin connection process, so they can (and should do it in that way) identify the (CPU context) interface and other context information, such as ID or version. After this recognize process plugins recognize (or do not recognize) supported CPU they can be connected with.
  - **Returns** - CPU context object

- *getDebugColumns*
  `public IDebugColumn` **getDebugColumns**`( )`
  - **Usage**
    * Gets columns in debug window. These columns will be used in the list in the debug window. Usually CPU uses these columns: ”breakpoint”, ”address”, ”mnemonics” and ”opcode”.

  - **Returns** - debug columns array

- *getDebugValue*
`public Object` **getDebugValue**`( int` **row**`, int` **col** `)`

  - **Usage**
    * Gets the value of a cell in debug window on specified position.
  - **Parameters**
    * `row` - cell's index from memory position 0 (not row in debug table)
    * `col` - column of the cell in in debug window table
  - **Returns** - value of the cell

- *getInstrPosition*
`public int` **getInstrPosition**`( )`

  - **Usage**
    * Get actual instruction position (before its execution). Can be said, that this method should return PC (program counter) register (if CPU has one).
  - **Returns** - memory position (address) of next instruction

---

- *getInstrPosition*
`public int` **getInstrPosition**`( int` **pos** `)`

  - **Usage**
    * Method compute address of an instruction that follows after instruction defined by given address. Main module uses this method to determine on which address should start next instruction in debug window. Several calls of this method make possible to create a list of instructions that begin on arbitrary address (debug window table).
  - **Parameters**
    * `pos` - memory position (address) of an instruction
  - **Returns** - address of an instruction followed by specified address

---

- *getStatusGUI*
`public JPanel` **getStatusGUI**`( )`

  - **Usage**
    * Gets CPU GUI panel. Each CPU plugin should have GUI panel that shows some important CPU status (e.g. registers, flags, run state, etc.) and allow to user perform some settings (e.g. set the frequency, etc.). This panel is located on right side in panel "emulation" in main module. CPU plugin should update the panel immediately after CPU state changes somehow.
  - **Returns** - status GUI panel (instance object)

- *initialize*
  `public boolean` **initialize**`( plugins.memory.IMemoryContext` **mem,** `plugins.ISettingsHandler` **sHandler** `)`
  - **Usage**
    - Perform initialization of CPU. This method is called after compiler successful initialization. Initialization process of CPU can be various, e.g. check for memory type, retrieve some settings from configuration file, etc.
  - **Parameters**
    - `mem` - memory context that this CPU should use. If CPU and memory aren't connected in abstract scheme, this will be `null`. Plugin should therefore check this variable and in the case of null memory and if CPU can't work without memory, plugin should display error message and then return false.
    - `sHandler` - settings handler object. CPU can use this for accessing/storing/removing its settings.
  - **Returns** - true if initialization was successful, false otherwise

---

- *isBreakpointSupported*
  `public boolean` **isBreakpointSupported**`( )`
  - **Usage**
    - Determine whether breakpoints are supported by CPU.
  - **Returns** - true if breakpoints are supported, false otherwise

---

- *pause*
  `public void` **pause**`( )`
  - **Usage**
    - Pauses the CPU emulation. If a thread was used for CPU execution and is running, then it should be stopped (destroyed) but the CPU state has to be saved for future run. CPU changes it state to "breakpoint".

- *reset*
  `public void` **reset**`( int` **startAddress** `)`
  - **Usage**
    - Perform reset of the CPU with specific starting address. This is used when program starting address is known. Otherwise it is used standard `Plugin.reset()` method
  - **Parameters**
    - `startAddress` -

- *setBreakpoint*
  `public void` **setBreakpoint**`( int` **pos,** `boolean` **set** `)`

  - **Usage**
    * Set/unset a breakpoint to specified memory position (address). It should be called only if breakpoints are supported (`isBreakpointSupported()`).
  - **Parameters**
    * `pos` - memory address where breakpoint should be set/unset
    * `set` - true if breakpoint should be set, false otherwise

---

- *setDebugValue*
  `public void` **setDebugValue**`( int` **row**`, int` **col**`, java.lang.Object` **value** `)`

  - **Usage**
    * Called when user sets a value to a cell in debug window. This method should ensure proper changes in CPU's internal state, caused by this set. The main module calls this method only if the cell in debug window is editable by user (`IDebugColumn.isEditable()`).
  - **Parameters**
    * `row` - cell's index from memory position 0 (not row in debug table)
    * `col` - column of the cell in debug window table
    * `value` - new value of the cell

- *setInstrPosition*
  `public boolean` **setInstrPosition**`( int` **pos** `)`

  - **Usage**
    * Set new actual instruction position (that will be executed as next). It can be said, that a parameter represents new value of PC (program counter), if CPU has one. Otherwise CPU should interpret the position in the right manner.
      This method is called by main module when user perform "jump to address" operation.
  - **Parameters**
    * `pos` - new address of actual instruction
  - **Returns** - true if operation was successful, false otherwise

---

- *step*
  `public void` **step**`( )`

  - **Usage**
    * Perform one step of CPU emulation. It means that one instruction should be executed. CPU state changes to state "running", then it executes one instruction, and then it should return to state "breakpoint" or "stopped". Correct timing of executed instruction isn't so important.

- *stop*
  `public void` **stop**`( )`

  - **Usage**
    * Stops the CPU emulation. If a thread was used for CPU execution and is running, then it should be stopped (destroyed) but the CPU state can be saved. CPU changes its state to "stopped" and main module should now forbid execution any of methods `step()`,`pause()`,`execute()` until user resets the CPU. Debug window in main module should be updated with saved CPU state.

3.1.2 INTERFACE **ICPUContext**

Basic interface for CPU context. The context is used by plugins, that are connected to CPU.
CPU plugins can extend this interface to their own (with some new methods) and then the programmer should make it to be public in order to other plugins could have access to it.
Extended context may have methods for e.g. connecting devices to CPU, interrupts implementation, etc.

3.1.2.1 DECLARATION

public interface ICPUContext **implements** plugins.IContext

3.1.2.2 METHODS

- *addCPUListener*
  `public void` **addCPUListener**`( plugins.cpu.ICPUContext.ICPUListener` **listener** `)`
  - **Usage**
    * Adds the specified CPU listener to receive CPU events from this CPU. CPU events occur when CPU changes its state, or run state. CPU state events don't occur if CPU is running, only happens with run state changes. If listener is `null`, no exception is thrown and no action is performed.
  - **Parameters**
    * `listener` - the CPU listener

- *removeCPUListener*
  `public void` **removeCPUListener**`( plugins.cpu.ICPUContext.ICPUListener` **listener** `)`
  - **Usage**

    * Removes the specified CPU listener so that it no longer receives CPU events from this CPU. CPU events occur when CPU changes its state, or run state. CPU state events don't occur if CPU is running, only happens with run state changes. If listener is `null`, no exception is thrown and no action is performed.
  - **Parameters**
    * `listener` - the CPU listener to be removed

3.1.3 INTERFACE **ICPUContext.ICPUListener**

The listener interface for receiving CPU events. The class that is interested in processing a CPU event implements this interface, and the object created with that class is registered with a CPU, using the CPU's addCPUListener method. When the CPU event occurs, that:

- if the event is CPU's state change, then object's `stateUpdated()` method is invoked.
- if the event is CPU's run state change, then object's `runChanged()` method is invoked.

3.1.3.1 DECLARATION

public static interface ICPUContext.ICPUListener **implements** java.util.EventListener

3.1.3.2 METHODS

- *runChanged*
  `public void` **runChanged** `( java.util.EventObject` **evt,** `int` **runState** `)`
  - **Usage**
    * Invoked when an CPU's run state changes.
  - **Parameters**
    * `evt` - event object
    * `runState` - new run state of the CPU

- *stateUpdated*
  `public void` **stateUpdated** `( java.util.EventObject` **evt** `)`
  - **Usage**
    * Invoked when an CPU's state changes. The state can be register change, flags change, or other CPU's internal state change.
  - **Parameters**
    * `evt` - event object

3.1.4 INTERFACE **IDebugColumn**

Interface that holds information about column in debug window.

3.1.4.1 DECLARATION

```
public interface IDebugColumn
```

3.1.4.2 METHODS

- *getName*
  `public String` **getName**`( )`
  - **Usage**
    - Gets name (title) of the column.
  - **Returns** - title of this column

- *getType*
  `public Class` **getType**`( )`
  - **Usage**
    - Gets java type of the column. Mostly the column type is `java.lang.String`, but for breakpoint columns should be used `java.lang.Boolean` class.
  - **Returns** - Java type of this column

- *isEditable*
  `public boolean` **isEditable**`( )`
  - **Usage**
    - Determines whether this column is editable by user. For example, mnemonics column shouldn't be editable (if CPU doesn't support assembly in runtime), but breakpoint cells should. If the column is editable, main module after editing the corresponding cell invokes `ICPU.setDebugValue` method and this method should take care of internal change in CPU.
  - **Returns** - true if column (with all its cells) is editable, false otherwise

Package plugins.compiler

Package Contents *Page*

4.1 Interfaces

4.1.1 INTERFACE **ICompiler**

This interface is the core for compiler plugin types. These plugins should implement this interface once and only once.

4.1.1.1 DECLARATION

public interface ICompiler
implements plugins.IPlugin

4.1.1.2 METHODS

- *compile*
  `public boolean` **compile**`( java.lang.String` **fileName**`, java.io.Reader` **in** `)`
  - **Usage**
    - Compile a file into output file. Output file name the compiler should derive from input file name.
  - **Parameters**
    - `fileName` - name of input file (source code)
    - `in` - `Reader` object of the document - source code.
  - **Returns** - true if compile was successful, false otherwise

- *compile*
  `public boolean` **compile**`( java.lang.String` **fileName**`, java.io.Reader` **in**`, plugins.memory.IMemoryContext` **mem** `)`
  - **Usage**
    - Compile a file into output file and into an operating memory. Output file name the compiler should derive from input file name.
  - **Parameters**
    - `fileName` - name of input file (source code)
    - `in` - `Reader` object of the document - source code.
    - `mem` - memory context object - it is used if compiler compiles the source into memory. Compiler should check this parameter for `null`.
  - **Returns** - true if compile was successful, false otherwise

- *getLexer*
  `public ILexer` **getLexer**`( java.io.Reader` **in** `)`
  - **Usage**
    - Get a lexical analyzer of the compiler. It is used by main module for syntax highlighting and of course in compile process by the compiler. For every call it should return new object (instance).
  - **Parameters**
    - `in` - `Reader` object of the document - source code.
  - **Returns** - lexical analyzer object

- *getProgramStartAddress*
  `public int` **getProgramStartAddress**`( )`
  - **Usage**
    - Gets starting address of compiled source. It is (or can be) called only after compilation process. It should return the first occurrence of compiled program. The word ”address” can be replaced by a term ”offset from 0”. One step in address has size of one byte. It means that return value should not be related to operating memory units and should not deliberate techniques used in operating memory, e.g. banking.
  - **Returns** - address of program’s first occurrence

- *initialize*
  `public boolean` **initialize**`( plugins.ISettingsHandler` **sHandler**`,`
  `plugins.compiler.IMessageReporter` **reporter** `)`
  - **Usage**
    - Perform initialization process of a compiler. This method is called immediately after plugins are loaded into memory.
  - **Parameters**
    - `sHandler` - settings handler object. Compiler can use this for accessing/storing/removing its settings.
    - `reporter` - object for reporting messages (warnings, errors, ...). This object is implemented in main module and is connected to text area in panel ”source code” in the main module. Plugin should use this.
  - **Returns** - true if initialization was successful, false otherwise

4.1.2 INTERFACE **ILexer**

Interface that implements lexical analyzer

4.1.2.1 DECLARATION

```
public interface ILexer
```

4.1.2.2 METHODS

- *getSymbol*
  `public IToken` **getSymbol**`( )`
  - **Usage**
    - Gets next lexical symbol from source code, from actual position. This is real implementation of lexical analyzer. After this call it should increase internal counters to next unread text (actual position, actual row, column, etc.)
  - **Returns** - next found token

- *reset*
  `public void` **reset**`( )`
  - **Usage**
    - Performs reset of the analyzer. Internal counters (actual position, actual column, row, etc.) should be cleared. Lexical analyzer should prepare itself for start from beginning of the document.

- *reset*
  `public void` **reset**`( java.io.Reader` **in**`, int` **yyline**`, int` **yychar**`, int` **yycolumn** `)`
  - **Usage**
    - Performs reset of the analyzer. Internal counters (actual position, actual column, row, etc.) should be cleared. Lexical analyzer should prepare itself for start from beginning of the document.
  - **Parameters**
    - `in` - `Reader` object of the document - source code.
    - `yyline` - from this line should lexical analyser start, usually 0
    - `yychar` - from this char should lexical analyser start, usually 0
    - `yycolumn` - from this column should lexical analyser start, usually 0

4.1.3 INTERFACE **IMessageReporter**

Interface for reporting messages while running compilation process. It is used for sending compiling messages to main module, e.g. warnings, errors, etc. These messages are showed in bottom text area in panel "source code" in the main module.

4.1.3.1 DECLARATION

public interface IMessageReporter

4.1.3.2 FIELDS

- public static final int TYPE_WARNING
  - –
- public static final int TYPE_ERROR
  - –
- public static final int TYPE_INFO
  - –

4.1.3.3 METHODS

- *report*
  `public void` **report**`( int` **row**`, int` **column**`, java.lang.String` **message**`, int` **type** `)`
  - **Usage**
    - Method reports some message to a main module with location information.
  - **Parameters**
    - `row` - row in the source code that is related to the message
    - `column` - column in the source code that is related to the message
    - `message` - message to report
    - `type` - type of the message (one of the `TYPE_WARNING`, `TYPE_ERROR`, `TYPE_INFO`)

- *report*
  `public void` **report**`( java.lang.String` **message**`, int` **type** `)`

- **Usage**
  - Method reports some message to a main module.
- **Parameters**
  - `message` - message to report
  - `type` - type of the message (one of the `TYPE_WARNING`, `TYPE_ERROR`, `TYPE_INFO`)

4.1.4 INTERFACE **IToken**

Interface that identifies a token.

4.1.4.1 DECLARATION

```
public interface IToken
```

4.1.4.2 FIELDS

- public static final int RESERVED
  - Token is a reserved word.
- public static final int PREPROCESSOR
  - Token is a preprocessor keyword.
- public static final int REGISTER
  - Token is a CPU register.
- public static final int SEPARATOR
  - Token is a separator (e.g. ' ', tab, '‘n', ...).
- public static final int OPERATOR
  - Token is a operator (e.g. +, -, *, /, ...).
- public static final int COMMENT
  - Token is a comment. Most assemblers used semicolon (";") as start of a comment.
- public static final int LITERAL
  - Token is a literal (e.g. number, string, char, ...).
- public static final int IDENTIFIER

  - Token is an identifier (e.g. name of variable, name of macro...).

- public static final int LABEL
  - Token is an label identifier.
- public static final int ERROR
  - Token is of unknown type.
- public static final int TEOF
  - Token represents end-of-file. This token should be the last found token.

4.1.4.3 METHODS

- *getColumn*
  `public int` **getColumn**`( )`
  - **Usage**
    - Gets 0-based column number of start of the token.
  - **Returns** - start column number of the token

- *getErrorString*
  `public String` **getErrorString**`( )`
  - **Usage**
    - Gets error string for this token. If the token is not errorneous, return empty string ("").
  - **Returns** - error string message for this token

- *getID*
  `public int` **getID**`( )`
  - **Usage**
    - Gets ID of the token. ID should be used for identifying not the type of the token (e.g. reserved word, etc.) but for concrete token of given token type got from `getType()` method. E.g. "mvi" is concrete token of type `RESERVED` and this method should return ID for "mvi" token. This is primary method for identifying tokens.
  - **Returns** - ID of the token

- *getLength*
  `public int` **getLength**`( )`
  - **Usage**
    - Gets length of the token.

  - **Returns** - length of the token

- *getLine*
`public int` **getLine**`( )`
  - **Usage**
    - Gets 0-based line (row) number of start of the token.
  - **Returns** - start line number of the token

- *getOffset*
`public int` **getOffset**`( )`
  - **Usage**
    - Gets 0-based offset from the start of the token.
  - **Returns** - starting offset of the token in the source code

---

- *getText*
`public String` **getText**`( )`
  - **Usage**
    - Gets textual representation of this token (token value).
  - **Returns** - token value

- *getType*
`public int` **getType**`( )`
  - **Usage**
    - Gets type of the token. Type is represented by pre-defined constants in this interface (e.g. reserved words, preprocessor, ...).
  - **Returns** - type of the token

- *isInitialLexicalState*
`public boolean` **isInitialLexicalState**`( )`
  - **Usage**
    - Initial lexical state represents positions in source code from where lexical analyzer can start parsing the code as it would parse from the start. Hence they are safe positions from which syntax highlighting can be reset, too.
  - **Returns** - true if token is in initial lexical state, false otherwise.

Package runtime

Package Contents *Page*

Classes

5.1 Classes

5.1.1 CLASS **StaticDialogs**

This class offers static methods that show some messages on the screen. Plugins should use these methods for displaying messages. Example of use: `StaticDialogs.showMessage("Hello, world!");`

5.1.1.1 DECLARATION

public class StaticDialogs
extends java.lang.Object

5.1.1.2 CONSTRUCTORS

- *StaticDialogs*
  `public` **StaticDialogs**`( )`

5.1.1.3 METHODS

- *getModelMinor*
  `public static int` **getModelMinor**`( )`
  - **Usage**
    - Get communication model minor version number. It's a part of its identification for plugins. Plugins can identify the model and if they are not compatible with it, they can raise an error.
  - **Returns** - minor version number

- *getModelVersion*
  `public static int` **getModelVersion**`( )`
  - **Usage**
    - Get communication model version number. It's a part of its identification for plugins. Plugins can identify the model and if they are not compatible with it, they can raise an error.
  - **Returns** - major version number

- *showErrorMessage*
  `public static void` **showErrorMessage**`( java.lang.String` **message** `)`
  - **Usage**
    - Show error message as `JOptionPane` dialog. Title will be ”Error”.
  - **Parameters**
    - `message` - error message to show

- *showMessage*
  `public static void` **showMessage**`( java.lang.String` **message** `)`
  - **Usage**
    - Show information message as `JOptionPane` dialog. Title will be ”Message”.
  - **Parameters**
    - `message` - information message to show

Package plugins.device

Package Contents *Page*

6.1 Interfaces

6.1.1 INTERFACE **IDevice**

Main interface that has to be implemented by device plugin.

Design of the interface supports hierarchical connection of devices. Devices identifies each other by context methods `getID()`, and `getHash()`. The connection request can be accepted or rejected if attaching device is/isn't supported. Devices that don't support any connection hierarchy, invoking their `attachDevice()` method, always return `true`.

6.1.1.1 DECLARATION

public interface IDevice
implements plugins.IPlugin

6.1.1.2 METHODS

- *attachDevice*
  `public boolean` **attachDevice**`( plugins.device.IDeviceContext` **male** `)`
  - **Usage**
    - Perform a connection of two devices. Male is a context of another device (that want to be connected into this one). Method **should carefully** check if male can be connected to this device, by recognizing its context. This is the only way how to ensure correctness of the connection.
  - **Parameters**
    - `male` - male-plug device context
  - **Returns** - true if connection process was successful, false otherwise

- *detachAll*
  `public void` **detachAll**`( )`
  - **Usage**
    - Detach all devices from this device. This method is invoked by main module in application closing process.

- *getNextContext*
  `public IDeviceContext` **getNextContext**`( )`
  - **Usage**

    - Get next context of this device. The device can have more than one context. In connection process, the main module asks for next device context for each connection case. If the device has only one context, it should return only this context for each call of this method.
  - **Returns** - next device context

- *initialize*
  `public boolean` **initialize**`( plugins.cpu.ICPUContext` **cpu,** `plugins.memory.IMemoryContext` **mem,** `plugins.ISettingsHandler` **sHandler** `)`
  - **Usage**
    - Perform initialization process of this device. It is called by main module after successful initialization of compiler, CPU and memory. Device should initialize itself besides other things also in a way of checking supported CPU and memory.
  - **Parameters**
    - `cpu` - context of a CPU. Device should check this for extended context. Will be `null` if a device is not connected to CPU.
    - `mem` - context of a memory. Device should check this for extended context. Will be `null` if a device is not connected to memory.
    - `sHandler` - settings handler object. Device can use this for accessing/storing/removing its settings.
  - **Returns** - true if initialization process was successful

---

- *showGUI*
  `public void` **showGUI**`( )`
  - **Usage**
    - Shows GUI of a device. Device don't have to have a GUI, but instead it should display information message.

6.1.2 INTERFACE **IDeviceContext**

Interface for basic context of the device. If device support more functionality than input or output, it should be extended (or implemented by an abstract class), and then make public.

6.1.2.1 DECLARATION

public interface IDeviceContext **implements** plugins.IContext, java.util.EventListener

6.1.2.2 METHODS

- *getDataType*
  `public Class` **getDataType**`( )`
  - **Usage**
    - Get the type of transferred data. As you can see, methods `in` and `out` use `Object` as the data type. This method should make the data type specific.
  - **Returns** - data type of transferred data

- *in*
  `public Object` **in**`( java.util.EventObject` **evt** `)`
  - **Usage**
    - Perform ”IN” operation, it reads data from this device. The device should return one byte of its input data. I/O operations are considered as events that occurred to this device.
  - **Parameters**
    - `evt` - event object
  - **Returns** - input data read from device

- *out*
  `public void` **out**`( java.util.EventObject` **evt,** `java.lang.Object` **val** `)`
  - **Usage**
    - Perform ”OUT” operation, it writes a data to this device. The device should accept one byte of the data that parameter `val` offers. I/O operations are considered as events that occurred to this device.
  - **Parameters**
    - `evt` - event object
    - `val` - data to be written to a device