Alecsis 2.3

the Simulator for Circuits and Systems

User's Manual

Laboratory for Electronic Design Automation
University of Niš, Faculty of Electronic Engineering
Beogradska 14, 18000 Niš, Yugoslavia

1/1998

Alecsis Manual

Željko Mrčarica Dejan Glozić
Vančo Litovski

Dejan Maksimović
Tihomir Ilić
Dragan Gavrilović

Laboratory for Electronic Design Automation
University of Niš, Faculty of Electronic Engineering

Beogradska 14, 18000 Niš, Yugoslavia

http://leda.elfak.ni.ac.yu/

1/1998

Foreword

Electronic circuit simulation constantly attracts attention of the scientific community. It is the ever-growing nature of electronic circuits that makes the simulation problem difficult. Algorithms and methods that proves to be efficient for circuits with 100,000 gates, run out of steam when faced with 1 million gates, while others cannot cope with heterogeneous circuits with micro-mechanical and/or optical devices and systems. It is well known that circuit simulation is very resource-intensive and that requirements for simulation of modern electronic circuits are always one step ahead of the state-of-the-art memory and CPU capabilities. Modern ASIC frequently contain analogue, logic, and even microelectromechanical subsystems. In addition, power electronic devices are integrated, driving capacitive as well as inductive loads. Self -heating effects and electrothermal simulation is becoming more and more important. To be able to handle an entire system, a modern simulator must have the ability to express all kinds of sub-systems in the most efficient way, with a desired level of accuracy.

Two main components are to be developed when simulation is considered: the language used for description of the system and simulation requirements, and the simulation engine itself. If we assume that non-electrical devices may be expressed using sets of nonlinear differential equations, the problem of modern simulation essentially requires two different algorithms - one for solving the sets of nonlinear differential equations (ordinary and/or partial) and another for discrete event simulation. It is therefore necessary to have a behavioural simulator which handles mixed-signal systems. Thus, analogue electrical and non-electrical portions of a system are analyzed using accurate and detailed, but quite slow and resource-intensive algorithms, while logic subsystems are expressed using logic values and states, with reverse accuracy and efficiency figures.

There are well established algorithms for simulation in both domains: analogue and digital. Furthermore, there are simulators that are widespread such as PSpice and HILO, to mention two only. What is new in modern developments, is integrated mixed-signal, mixed-level and mixed-domain simulation. Such a simulator is Alecsis 2.3, described in this manual. In addition, a hardware description language AleC++ was developed strongly dedicated to simulation. For that reason, Alecsis 2.3 is an integrated language-simulator system which is able to solve practically all simulation problems arising in electronics and neighbouring areas. Before conclude this foreword the importance of HDLs will be emphasized, so that a short state of the art will be given and AleC++ will be generally outlined.

Hardware description languages using programming language constructs have long been in use in the digital domain. Programming languages have been directly applicable to the task of behavioural description of digital systems as the behaviour of digital systems is naturally expressed in algorithmic form. In the past 30 years, over 200 different languages or environments, each presenting its own conceptual approach to simulating the given problem, have been published. With the introduction of VHDL (Very high speed integrated circuit Hardware Description Language) thing were changed dramatically. VHDL not only became a standard language to describe digital models and behaviour but it is now a basis for development of design tools including synthesis, documentation, graphic representation and so on. Its fundamental philosophy is ADA based. In the same time an additional standard was adopted the so-called Verilog language based on the C language. These two standards provided users with a richer set of tools to produce more reliable models and suppliers with a more rewarding market to explore. Of course, an oft-cited drawback of standards is their tendency to embody the lowest common denominator or to be outmoded by the time they are finalized. But this is certainly not exactly the case when VHDL is considered: it has not only helped to create a competitive digital simulation market but also has forced synthesis and other nonsimulation tools, all with high degree of interoperability. Both VHDL and Verilog now provide a common format for movement toward synthesizers and correct-by-construction tools.

Analogue simulation tools primarily began with early circuit simulation tools, with SPICE being the mainstay of the CAD tools used in analogue IC design process. The newer generation of simulation tools includes modelling at higher levels, and analogue hardware description languages. Modelling techniques such as macromodelling, which earlier were centred around the generation of equivalent circuit models, have now evolved to higher-level analytical models using analytical techniques and symbolic analysis. These higher level models are usually encoded in AHDLs (Analogue Hardware Description Languages). The newer generation of circuit simulators can simulate the mixed-circuit descriptions that combine circuit-level models and higher-level models. Still, among the analogue design bottlenecks one can include:

- the problem of large scale analogue circuit simulation (the performance of digital circuits, having in mind parasitic effects, cannot be evaluated without analogue simulation);
- the simulation of mixed analogue-digital environments which becomes very important in modern telecommunication systems;
- modelling and simulation across various domains (electrical, mechanical, thermal, acoustical, optical, working together in modern robot and medical applications);
- analogue HDL technology (the lack of standard analogue HDL) and so on.

Two early examples of commercial analog HDLs are MAST from Analogy Inc., and HDL-A distributed by Mentor Graphics Corp. A later entry into the proprietary analog HDL marketplace is SpectreHDL from Cadence Design System Inc. The common characteristic of all this languages is relation to a simulator which was developed earlier so becoming some kind of constraint to the language.

The three current standardization efforts that are underway tend to become technology and simulator independent which is similar to the digital counterparts. Chronologically, the first is the Mimic HDL or MHDL sponsored by MDHL Study Group, a part of the IEEE Standards Coordinating Committee – 30 (SSC-30). It was followed by VHDL-A (or VHDL-AMS), developed by the IEEE 1076.1 committee as the analogue extension of VHDL. The most recent is Verilog-A, proposed analogue extension to a co-standard IEEE Verilog language.

New HDL for analogue and mixed-mode descriptions are in development at several universities in parallel. These address specific topics such as verification orientation or component-simulation orientation involving constructs convenient for such system description. Microelectromechanical systems, for example, demand partial differential equation solving, which is not the case in electronic circuit simulation where ordinary nonlinear equations are solved. Having in mind complex interaction which are present in mixed-signal, mixed-level, and mixed-domain designs one intensively considers the object-orientation in both entity overloading and hierarchical modelling point of view.

Object-oriented programming (OOP) is both a general methodology or a way of thinking, and a tool for programming. It is possible to design and write programs based on OOP ideas without any specific OOP language, but a good object-oriented language directs and supports good programming practice.

OOP features can be implemented in any programming language, but some languages are more suitable and flexible than others. The objects serves as an abstraction mechanism where the implementation details are hidden so that the user can rely on a systematic and fairly simple interface between the objects and the external world. The most common features in object-oriented languages are:

- ♦ Definition of an object class as a general model or template for the instance objects of the class.
- ♦ Hierarchical inheritance of properties (slots and methods) from a superclass to a more specific subclass makes the object-oriented programs systematic and compact by modular construction and code sharing.
- ♦ Instances of an object class can be created and deleted at runtime.
- ♦ Computation is localized into the objects by defining methods or functions common to a class so that all instances of a class behave in a specific way. These methods functions also form a communication protocol and abstraction mechanism that hides the implementation details.
- ♦ A process feature of objects, i.e. defining objects as parallel processes, is useful if concurrent communication between objects is needed.

The most outstanding advantages gained by using object-oriented programming are:

- – Natural and clean representation of real world objects.

- Highly modular structure of programs and avoidance of many explicit branching control structures (if then, case, etc.).
- Good reusability of software constructs and modules.
- Easy reconfiguration and maintenance of programs.
- Compactness of programs due to code sharing by inheritance.

Having all this in mind, a language named AleC++ dedicated to simulation purposes was developed as a superset of C++. In addition to the general advantages of OOP mentioned above special constructs were introduced for both control the simulation process, and description of the system to be simulated. One may now use model class inheritance (very successful for hierarchical modelling in IC design environment), entity overloading (used in logic and mixed-signal simulation), clone operation (convenient for repetitive structure description), and many more. All together it may look a bit complicated but it offers much. The reader is advised to look for the introductory booklet "Alecsis the simulator" first. It will give him a glimpse over the whole system enabling better understanding of this Manual.

Contents

1. Introduction 1

2 Lexical basis of AleC++ 3

2.1. Blank space 4

2.1.1. Comments 4

2.1.2. Line connections 5

2.2. Symbols 5

2.2.1. Key words 5

2.2.2. Identifiers 6

2.2.3. Constants 6

2.2.3.1. Integer constants 7

2.2.3.2. Real constants 7

2.2.3.3. Character constants 9

2.2.3.4. Index (enumeration) constants 9

2.2.4. Operators 11

2.2.5. Separators 11

2.3. Preprocessor 12

3. Grammar of AleC++ 13

3.1. Declarations 13
3.1.1. Basic data types 14
3.1.2. Allocation method 15
3.1.2.1. auto 16
3.1.2.2. static 16
3.1.2.3. extern 16
3.1.3. Declaration of new types 16
3.1.4. Address alignment 17
3.1.5. Structures 17
3.1.6. Enumeration type 18
3.1.7. Multidimensional variables 19
3.1.8. Initialization 19
3.1.9. Declarations of functions 20
3.2. Definitions 21
3.2.1. Definition of functions 22
3.3. Expressions 23
3.4. Commands 25

4. Object-oriented programming 27

4.1. Constant variables 28
4.2. References 28
4.2.1. Formal references 29
4.2.2. Local references 29
4.3.3. Reference-returning functions 29
4.3. Function overload 30
4.4. Default values of function parameters 30
4.5. Inline functions 31
4.6. Functions with variable number of arguments 31
4.7. Visibility area resolution operator 32
4.8. Classes 32
4.8.1. Access to class members 33

4.8.2. Declaration and definition of methods 33
4.8.3. Keyword this 34
4.8.4. Static methods and class members 35
4.8.4.1. Static members 35
4.8.4.2. Static methods 36
4.8.5. Class friends 36
4.8.5.1. Friendly functions 36
4.8.5.2. Friendly classes 37
4.9. Constructors and destructors 37
4.9.1. Constructors 38
4.9.2. Destructors 39
4.10. Operator overload 39
4.10.1. Global level overload 39
4.10.2. Overload using methods 40
4.11. Overload of operator = ... 41
4.12. Overload of implicit conversions 41
4.13. Dynamical allocation of memory 42
4.13.1. Allocation - operator new 42
4.13.2. Deallocation - operator delete 43
4.14. Inheritance 43
4.14.1. Inheritance and rules concerning access rights 43
4.14.2. Access to members in the hierarchy bearing the same name 44
4.14.3. Virtual base classes 44
4.14.4. Construction and destruction of derived classes 45
4.15. Virtual functions 46
5. Basic simulation in Alecsis 48
5.1. Module concept 49
5.2. Link 50
5.3. Module declaration 51
5.4. Module definition 52
5.4.1. Declarative part 53
5.4.2. Structural part 54
5.4.3. Functional part -- action block 56
5.4.4. Modelling of parallel processes 58
5.4.5. Variable number of action parameters 61

5.5. Implicit declaration of components ... 62
5.6. The root module ... 63
5.6.1. Print control -- command plot ... 63
5.6.2. Timing control ... 67
5.6.3. Simulation options ... 69
5.6.3.1. Control of simulation time (numerical integration) ... 69
5.6.3.2. Control of convergence (iterative process) ... 73
5.6.3.3. Control of system of equations solver ... 77
5.6.3.4. Control of models ... 77
5.7. Model cards ... 78
5.7.1. Model cards as static objects ... 79
5.7.2. Syntax of model card ... 81
5.8. Modules with variable structure -- clone and allocate ... 83
5.9. Visibility area (name masking) ... 83
6. Digital simulation in Alecsis ... 84
6.1. Alecsis support of digital simulation ... 84
6.1.1. Systems of states ... 85
6.1.2. Truth tables ... 86
6.1.3. Overloading of operators ... 86
6.1.4. Overloading operators for vectors ... 87
6.1.4.1. Operator lengthof ... 87
6.1.4.2. Buffers for temporary solutions ... 87
6.2. Synchronization of digital processes ... 89
6.2.1. Interpretation of signals in expressions ... 89
6.2.2. Conversion of link type ... 91
6.2.3. Conditional process suspension -- command wait ... 91
6.2.4. Predefined signal attributes ... 92
6.2.5. Signal assignment -- operator '<-' ... 93
6.2.5.1. Drivers ... 93
6.2.5.2. Delay models ... 94
6.2.5.3. Resolution of conflict on the bus (resolution funciton) ... 95
6.2.5.4. Driver initialization ... 97
6.2.5.5. Complex delay ... 98
6.3. User-defined signal attribute -- operator '@' ... 98
6.4. Leaving out actual signals -- void ... 100
6.5. Variable number of formal signals -- operators '$' and '$$' ... 100
6.6. Variable number of action parameters ... 101

6.7. Array of components -- commands clone and allocate 101
6.8. Structural aspect of digital circuits 102

7. Analogue simulation in Alecsis 103

7.1. Built-in analogue models 104
7.1.1. Resistor, capacitor, inductor, and ideal sources 104
7.1.2. Built-in signal generators 105
7.1.2.1. Piecewise linear signal generators 105
7.1.2.2. Sinusoidal signal generators 107
7.1.3 Controlled sources 107
7.1.4. SPICE-compatible nonlinear components 108
7.1.4.1. Diode 108
7.1.4.2. MOS transistor 108
7.1.4.3. Bipolar junction transistor 109
7.1.4.4. JFET 109
7.1.5. Ideal switch 109
7.2. Structural modelling 112
7.3. Combined structural-functional modelling 112
7.3.1. Time-dependent linear models 114
7.3.2. Nonlinear models 115
7.3.3. General nonlinear sources (automated linearization) 118
7.3.3.1. Nonlinear current generator -- nlcgen 119
7.3.3.2. Nonlinear voltage generator -- nlvgen 122
7.3.3.3. Nonlinear equation -- nlgen 123
7.3.4. Virtual synchronization of processes 123
7.4. Functional modelling -- eqn statement 124
7.4.1. Simple eqn statement 126
7.4.2. Numerical integration in eqn statement (ddt, d2dt2, idt) 128
7.4.3. Through eqn statement 130
7.4.4. Across eqn statement 132
7.5. Appointed simulation in a time-instant -- breakpoint 134
7.6. Returning the link using name of a module 135
7.7. Variable number of action parameters 137
7.8. Modules with variable structure -- clone and allocate 137

8. Hybrid simulation in Alecsis 141

8.1. Implicit converters of link aspects 142
8.1.1. A/D conversion 143
8.1.2. D/A conversion 145
8.2. Converter declaration 147
8.2.1. Converter declaration for the whole circuit 147
8.2.2. Converter declaration for module 148
8.2.3. Converter declaration for formal signals 148
8.2.4. Organization of class hierarchy for digital model classes 149
8.3. An example of hybrid circuit simulation 151

Appendix 1 **Alecsis installation and use 154**

A1.1. Alecsis installation 154
A1.1.1. Paths for UNIX shell 155
A1.1.2. Compiling Alecsis source code 156
A1.1.3. Compiling Alecsis standard libraries 156
A1.2. Alecsis use 157
A1.2.1. Program call from the command line -- command options 157
A1.2.2. File name extensions 160
A1.2.3. Listing of libraries in the source file -- command library 160
A1.3. Overview of Alecsis versions 161

Appendix 2 **Alecsis standard libraries 162**

A2.1. alec.h 162
A2.2. ctype.h 165
A2.3. bit.h 166
A2.4. gnulib.h 167
A2.5. math.h 170
A2.6. unisdt.h 171
A2.7. varargs.h 172

Appendix 3 **Syntax of AleC++ 173**

Appendix 4 **AleC++ assembler** **198**

A4.1. Operands in assembler instructions 199
A4.2. Assembler instructions 200
A4.2.1. Instructions of Alecsis virtual processor 201
A4.2.2. Instructions of Alecsis virtual coprocessor 202
A4.3. Conventions on passing parameters to functions 203

Appendix 5 **Model card parameters for built-in components** **208**

A5.1. Diode model card parameters 208
A5.2. MOSFET model card parameters 209
A5.2.1. MOSFET level 1, 2 and 3 parameters 210
A5.2.2. BSIM parameters (level 13) 211
A5.3. BJT model card parameters 213
A5.4. JFET model card parameters 215

Appendix 6 **Analogue simulation examples** **217**

A6.1. Electronic models 217
A6.1.1. Resistor 218
A6.1.2. Capacitor 219
A6.1.3. Example 1 220
A6.1.4. Inductor 222
A6.1.5. Example 2 223
A6.1.6. Current source 225
A6.1.7. Voltage source 226
A6.1.8. Diode 227
A6.1.9. Example 3 229
A6.1.10. Sinusoidal voltage generator 231
A6.1.11. Example 4 232
A6.1.12. Triangular voltage generator 233
A6.1.13. Example 5 234
A6.1.14. Pulse voltage generator 235

A6.1.15. Example 6 ... 236
A6.1.16. Step voltage gnerator ... 237
A6.1.17. Example 7 ... 238
A6.1.18. Polynomial voltage generator ... 239
A6.1.19. Example 8 ... 240
A6.1.20. Submodule (subcircuit) example ... 241
A6.2. Mechanical models ... 242
A6.2.1. Oscillating mass ... 242
A6.2.2. Example 1 ... 244

Appendix 7 **Alecsis library manager -- alm ... 245**

A7.1. Handling of libraries in Alecsis ... 246
A7.2. Capabilities of alm ... 246
A7.3. Commands line options ... 248
A7.4. alm commands ... 249
A7.4.1. List of alm commands ... 249
A7.4.2. Overview of alm commands ... 249

Appendix 8 **Waveform display program -- agnu ... 256**

A8.1. agnu command line options ... 257
A8.2. agnu commands ... 257

Appendix 9 **Postprocessors nrl and nzd ... 258**

Appendix 10 **PSpice to Alecsis converter ... 261**

A10.1. Why PSpice2Alecsis conversion? ... 261
A10.2. Use of PSpice2Alecsis ... 262
A10.2.1. Analogue circuit conversion ... 263
A10.2.2. Digital circuit conversion ... 264
A10.2.3. Hybrid circuit conversion ... 265

A10.3. Conversion of PSpice commands with examples 265
A10.3.1. Conversion of components 265
A10.3.2. Conversion of simulation control statements... 278
A10.4. Limitations of PSpice2Alecsis converter 283
A10.4.1. Limitations of circuit topology description 283
A10.4.2. Limitations of commands for simulation control....................................... 284
A10.4.3. General limitations.. 285

1. Introduction

Alecsis 2.3 is the newest version of the integral, hybrid simulator developed in the Laboratory for Electronic Design Automation (LEDA) of the Faculty of Electronics, University of Niš. The name itself is an acronym of **A**nalogue and **L**ogic **E**lectronic **C**ircuit **SI**mulation **S**ystem. Alecsis 2.3 is an integrated simulator because it represents the coupling of the interpreter and the linker with the simulation engine, and the connection between the mechanisms for analogue and digital simulation. The following chapters will show that the circuit description syntax of the analogue and digital portion varies insignificantly across both types due to their inseparability.

All modern hardware description languages (HDLs) use one of the standard programming languages as a basis. Programming language C was chosen as the basis for Alecsis 1.1 due to its widespread acceptance. During the development of the next version (Alecsis 2.1) object-oriented programming capabilities were added through the use of C++ constructs. Version 2.3 was released as an improved, more effective version of Alescis 2.1. HDL used in Alecsis 2.3 is therefore a superset of C++ (the same way C++ is a superset of C.) The authors strive to emphasize the details which differ from the rules of C, or C++. If there exists ambiguity beyond these details any handbook on C, or C++ will help.

Alecsis is a package consisting of language interpreter/compiler and the simulation engine. In the text the package will be referred to as Alecsis 2.3, or just Alecsis. In order to differentiate between the abstract notion of a language and the functional programming package the language itself will be referred to as AleC++.

This Manual is split in eight chapters and seven appendices. The second and third chapter define the lexical and grammatical rules of AleC++. Object-oriented construct of the language are introduced in the fourth chapter. Basic simulation capabilities are presented in the fifth chapter, and their application in the in the analogue and digital domains in the sixth and seventh chapters, respectively. Finally, the application to the hybrid circuits is given in the eighth chapter. Application and installing of the whole package, as well as the content of the standardized libraries is given in the Appendix A. Appendix B offers the complete syntax of AleC++. Commands of the assembly language of the virtual processor used in the simulator are explained in Appendix C. Appendix D describes ALM (Alecsis Library Manager) which enables easy handling of the user-defined libraries. Appendix E develops the description of the graphical postprocessor Agnu 1.1, used in conjunction with a shareware graphical

display program Gnuplot. The graphical display of the simulation results using Agnu is possible during the simulation run, too.

The following terminology will be used in Appendix B and the parts of the text concerning the syntax rules of AleC++.

- Syntax categories will be defined using the cursive, e.g.

  global_definition:

  global_variable

  function

- Listing of categories using ‘:’ has the meaning “one of.” If a category may, or may not be included, it will be marked using ‘<‘ and ‘>‘:

  ptr_operator:

  * *<cv_qualifier>*

- If multiple options were to appear in single row the will be emphasized using the note “one of:”

  decimal_digit: *one of*

  0 1 2 3 4 5 6 7 8 9

- Bold symbols represent the reserved words:

  cv_ qualifier :

  const

  volatile

2. Lexical basis of AleC++

The basic role of the interpreter is to analyze a string of ASCII characters, group them in symbols and check if the combination of those symbols agrees with the rules of the language in question. Lexical analyzer does the job of grouping up the symbols. AleC++ has inherited the lexical rules of C++ for the most part; the differences will be documented in detail.

The names of Alecsis input files are arbitrary (the name length is determined by the specific operating system,) but they have to have the extension `.ac`. Extension `.hi` is also allowed for compatibility with version 1.0. File name extensions are the following:

`ac` - Alecsis input file
`hi` - Alecsis 1.0 input file (accepted by newer versions, too)
`h` - Alecsis header file (as in C/C++)
`ar` - Alecsis results (Alecsis output file, Agnu input file)
`ao` - Alecsis object-code file (compiled input file)
`as` - Alecsis assembly language file (created by compiler using option `-S`)
`aa` - Alecsis library
`stat` - Alecsis statistics file (creted when command option `-stat` is used)

The preprocessor processes the file by analyzing the lines beginning with the special character '#', and develops all preprocessor macros in the text. The interpreter in fact analyzes the results supplied by the preprocessor (temporary file,) and not the original text.

2.1. Blank space

Blank space, or blank text is the text which does not produce any effect since the interpreter ignores it. It is used for symbol separation, increased readability of the text and documentation purposes. Since AleC++ is a

superset of C++, it is a free format language (all construct can be extended to an arbitrary number of lines.) Empty character string ' ', horizontal tabulator, and a new line are examples of blank space. These characters are not treated as blank space only if they are in between characters ' ' or " " , e.g. if they are a part of character, that is string constants. Beside the above mentioned characters the interpreter treats comments as blank space, as well.

2.1.1. Comments

Comments allow for the documentation of the text. AleC++ supports three types: basic, line and SPICE comments.

Basic comments are an arbitrary text bounded by `/*` and `*/` (C comments.) They can be arbitrarily long, but cannot be nested.

```
/* this is a one-line comment */

/****************************************
 *   this is a multiple-line comment      *
 ****************************************/

/*   /* this is an error */ */
```

Line comments are inherited from C++, and last until the end of the current line. The text right of character `//` is considered a comment.

```
// this is a line comment
```

The third type of comments (SPICE) are used in a limited number of cases. Since Alescis supports analogue device models of SPICE simulator, AleC++ has a shortcut for the users using libraries of model cards for mentioned components:

```
spice {
* SPICE syntax is valid in between the characters { and } - this
* is a SPICE comment
.model mn nmos ( level=1 vto=0.7v )
}
```

After reserved word spice inside parentheses { and } only SPICE lexical rules are valid. It means that everything starting with '`*`' is a comment. The lexical analysis goes until the end of the line, which can be continued if '+' sign is in the first column. Characters '`(`' and '`)`' are ignored. In the end the line has to begin from the first column. As much as this rule are cumbersome, they allow direct use of hundreds or even thousands of lines of text using SPICE cards, and can significantly shorten the time needed to model the same elements using Alecsis.

2.1.2. Line connections

Sometimes it may be necessary to define a string longer than the length of one line. If character '\' is placed at the end of the first line, it will be ignored along with the end of the line. This results in the merger of two lines, i.e. the string continues starting with the beginning of the next line. Note that standard C offers this option, as well. Using this method, constant strings can be defined across many lines:

```
"this string spans across \
two lines"
```

2.2. Symbols

Now that we have defined blank space the only thing left are symbols. Symbols in AleC++ are:

- key words
- identifiers
- constants
- operators
- separators

2.2.1. Key words

Key (reserved) words have special meaning and are not to be used outside their definition (except within string constants.) Key words in AleC++ represent a amalgam of key words from C++ and a smaller number of ones created for hardware description. Basic (C) key words are shown in Table 2.1.

Table 2.1: Key words in C.

auto	break	case	char	const	continue	default
do	double	else	enum	extern	float	for
goto	if	int	long	register	return	short
signed	sizeof	static	struct	switch	typedef	union
unsigned	void	volatile	while			

In addition to this ones C++ introduced 11 more key words (Table 2.2) Since AleC++ supports C++ syntax for the most part, these are a part of AleC++ syntax, too:

Table 2.2: Key words in C++

class	delete	friend	inline	new	operator	private
protected	public	this	virtual			

AleC++ has key words used for electronic circuit description. These key words are shown in Table 2.3.

Table 2.3: Key words used in AleC++ only.

action	allocate	asm	after	bjt	capacitor	cccs
ccvs	cgen	charge	clone	conversion	current	ddt
diode	eqn	flow	idt	in	inout	implicit
inductor	jfet	lengthof	library	model	module	mosfet
nlcgen	nlgen	nlvgen	node	now	out	options
plot	process	resistor	root	signal	sweep	temp
timing	transport	vccs	vcvs	vgen	vsin	vpwl
wait						

2.2.2. Identifiers

Identifiers are symbols; names of variables, functions, markers, elements, etc. The names can consist of an arbitrarily long number of characters `a-z`, `A-Z`, and `0-9`, as well as '`_`'. There are two rules to honour:

Identifier cannot begin with a digit;

Number of characters may vary from implementation to implementation (it depends if the identifier appears in the file system, where names are limited to 8-32 characters.) The current version of Alecsis provides for 255 characters.

Examples of identifiers are:

```
i   counter   i1   i123_a   __fetch2   VeryLongButCorrectIdentifier
```

AleC++ is *case-sensitive*, i.e. capital and small letters differ. The exception to the rule is SPICE environment, since SPICE is not case-sensitive.

2.2.3. Constants

Constants store fixed values of numbers, signs, or strings. AleC++ supports 4 types of constants: integer, real, index, and character.

2.2.3.1. Integer constants

Integer constants can have in decimal, octal, or hexadecimal format. Decimal constants represent a sequence of digits `0-9`, bearing in mind that the first digit cannot be `0`. The length restrictions depend upon the actual implementation, but most UNIX computers are AleC++ integers represented stored using 4 bytes. An example of a decimal constant is:

`1`

`12`

`1279`

but not:

`037` (octal)

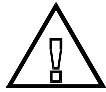
`0x22` (hex)

`-2` (expression)

Decimal constants larger than 2 147 483 648 cause error.

Octal constants consist of a sequence of digits `0-7`, bearing in mind that the first digit cannot be `0`. The error will occur if the octal number is greater than `017777777777`.

Hexadecimal constants need to begin with `0x`, or `0X`. They consist of sequence of digits `0-9` and characters `a-f`, or `A-F`. A hexadecimal constant cannot be larger than `0x7fffffff`.



AleC++ does not support unsigned types. ANSI-C suffix type `u` or `U` (from `unsigned`) are not allowed. Since the number of bytes occupied by the types `short`, `int`, and `long` is identical (4 bytes), suffixes `I` and `L` are not supported either. All integer operations in AleC++ are performed as `signed long`. Reader needs to note that the constant `3u` does not mean `(unsigned)3`, but rather `3.0e-6`, because suffix `u` in AleC++ means *micro*.

2.2.3.2. Real constants

The representation of real constants is the same as in C, or C++. These are a few examples of real constants:

```
1.
1.2
.2
.2e-3
1e12
0.22334
1E12
```

AleC++ introduces a concept of *units*, not unlike similar hardware description languages. To simplify writing of physical constants, one can use suffices that denote thousand times smaller or bigger units. Following examples are valid:

`f` or `F` - 1e-15

`p` or `P` - 1e-12

`n` or `N` - 1e-9

`u` or `U` - 1e-6

`m` (without `M`) - 1e-3

`k` or `K` - 1e3

`M` - 1e6

`g` or `G` - 1e9

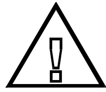
`t` or `T` - 1e12

Note: An integer constant becomes a real constant if followed by one of the shown suffixes. It means that `1k` means the same as `1.0k` or `1e3`.

A constant can have a user-defined suffix consisting of alphabets `a-z`, `A-Z`, and/or _. The purpose of that suffix is to clearly define physical units of measurement and it is ignored in computing. It follows that the constants:

```
1k    1kohm    1ohm    1.23pF    33MHz    33cycles
```

are written correctly. An integer constant without the suffix, and with a suffix that is not an unit remains an integer constant.



If you write `1Pa` for a pressure of 1 Pascal, AleC++ will understand it as number of 1.e-12, as P is understood as multiplication with 1.e-12. For that reason, be very careful when writing suffixes for physical constants, or better use only suffixes for multiplying (kilo, milli, micro) and omit the physical unit itself.

It should be noted that SPICE units suffixes are appropriate for text marked by the key word `spice`. More information can be obtained from any of the manuals covering SPICE program.

2.2.3.3. Character constants

A character bounded by the apostrophes is a character constant. If that character cannot be displayed or has a special meaning, the *escape* sequence can be used:

```
'c'     'a'     '+'     '\n'     '\\'     '\007'     '\t'
```

Strings are sequences of characters bounded by " ":

```
"string"
"one more"
"string with the escape character for a new line \n"
```

Rules which apply in ANSI-C, or C++ , apply in AleC++. Note: AleC++ merges all strings separated by a blank space (ANSI-C), e.g.

```
"first and "   "second"
```

merge into

```
"first and second"
```

2.2.3.4. Index (enumeration) constants

As in C, enumeration constants are declared using the key word `enum`:

```
enum Bool { False, True };
```

Constant `False` has the value of `0`, and constant `True` is `1`. The values increase by one starting from `0`, as the new symbols are added. If this setup is not satisfactory in a special case a direct intervention is possible:

```
enum Bool { False, Fatal = 0, True, OK = True };
```

Symbols without the initial value are assigned the value 1 greater than the former value. The initial value has to be constant, or already defined index symbol.

Enumeration constants are a part of C and C++, but with some changes:

- In C, index symbols are accessible from all expressions of same or narrower area of visibility. **Index symbols in AleC++ are accessible if, and only if the enumeration group can be determined from the context.** This allows for the same symbols to be used within two, or more enumeration groups; an ability not found in other languages. An example of this is:

```
Bool status1 = True, status2 = False;
```

but not:

```
int status3 = OK;
```

since one cannot determine from the context which group is involved (`status3` is `int`)interpreter will report an error. This modification was necessary for logic simulation, as enumeration constants are used for logic states. One state can be found in more than one set of possible logic states.

- **Enumeration symbols in AleC++ can be character constants,** since we can determine from the context if the constant is an enumeration one, and which group it belongs to. They do not have ASCII values, but values according to their place in the group. If the interpreter cannot determine from the context if it is an enumeration constant, it will treat it as a character constant.

```
enum digital3 { '0', '1', 'x' };         // index 0, 1, 2
enum digital4 { '0', '1', 'z', 'x'};     // index 0, 1, 2, 3

char c = 'x';            // character - ASCII values
digital3 d3 = 'x';       // enum digital3 - value 2
digital4 d4 = 'x';       // enum digital4 - value 3
```

Enumeration groups of characters are the basis for design of state systems for modelling of digital hardware. Since the symbols are valid only within the group, and bear no influence on other groups, it is possible to form an unlimited number of states, which have the same states (it is realistic to expect states '0' ,'1', and 'x' to repeat often.)

- In AleC++, an **enumeration string** can be defined. It does not differ from the common one, and the recognition by the interpreter is done in that usual way, as is the case with the individual constants. Enumeration strings can consist of symbols defined in the appropriate enumeration group.

```
char *s = "string character"; // common string
digital3 *d3s = "0001x1x1";   // enumeration string - digital3
digital4 *d4s = "zzzz1101";   // enumeration string - digital4
digital4 *d4e = "0101xxaa";   // error - 'a' is not in the group
```

Common strings end with the character '`\0`', which is (n+1)st character of a string with length n, although that character is not displayed. Enumeration strings do not have that character at the end since the first symbol in the enumeration group is 0. **To determine length of an enumeration string, one has to use new command** `lengthof` to be explained in the following chapters.

- Longer enumeration strings can be a reading challenge, i.e.:

```
"0001011100xx0011"
```

This string representing a 16-bit word would be readily understandable if bytes, or even nibbles were separated. To that goal AleC++ introduces enumeration separator, a common non-indexed character constant, skipped in enumeration strings:

```
enum digital3 { '0', '1', 'x', 'X' = 'x', ' ' = void, '_' = void };
digital3 es1[] = "0001011100xx0011"; // string without the separator
digital3 es2[] = "0001_0111 00XX_0011"; // string with separators
```

This example shows that multiple separators can be introduced, initialized as **void**. The value of both `lengthof(es1)` and `lengthof(es2)` would be 16, since **the separator does not affect the length of the string**. The first symbol after the separator assumes the index value one larger than the index value of the symbol before the separator **since the declaration of the separator does not affect indexing**.

Note: The example above shows case-insensitivity (both 'x' and 'X' have value 2).

2.2.4. Operators

Operators are symbols that indicate arithmetic, logical, and other operations over symbols-operands. AleC++ supports existing C, and C++ operators, and defines some new ones. These are:

`~&`	binary NAND
`~|`	binary NOR
`~^`	binary XNOR
`<-`	signal assignment
`$`	direct access to formal signals
`$$`	total number of formal signals
`@`	attribute, partial differentials
`ddt`	first time derivative
`d2dt2`	second time derivative
`idt`	time integral
`sdt`	second partial time differential $\left(\frac{d^2 f(x)}{dtdx}\right)$

The new operators were introduced to satisfy the needs for simulation and functional modelling. While the first three are a simple negation of existing ones, the fourth operator makes the basis for modelling of the communication between parallel processes (you can read more in the chapter on digital circuits modelling.) The rest of the operators are going to be discussed in the text that follows.

2.2.5. Separators

The list of the separators will be introduced now while the detailed explanation of their usage will be left for following chapters:

```
[ ]  { }  ( )  : ; , ... #
```

2.3. Preprocessor

Preprocessor is a separate part of the interpreter, which analyses the text, and creates temporary file. Preprocessor commands differ from the others by the symbol “#” situated in the first column. Preprocessor can define macros, with or without the parameters, include other files, or control the parts of the text to be interpreted. Preprocessor of AleC++ supports the standard C preprocessor partially in the following directives:

```
#define
#include
#ifdef
#ifndef
#else
#endif
```

These directives are fully supported, and can be used without limitations. In case of need other directives will be included in the follow-up versions of AleC++.

The `include` command functions with the names of files between characters " and ", or in between characters < and >. Files with names given between characters < and > are searched for in the system directory, defined by the system variable ALEC_HOME (see the installation procedure). Files with names given with " and " are searched for in the current directory.

3. Grammar of AleC++

Previous chapter dealt with the way characters form symbols according to lexical rules. Grammar of a language determines a set of legal ways those symbols can be combined. The grammar of AleC++ encompasses basic grammar (discussed in this chapter), the grammar of object programming, and constructs for simulation description (following chapters).

The overview of the grammar of AleC++, which will be given will be rather brief since the authors assume the reader already knows C/C++. The instructions given will be concentrated on the constructs of AleC++; if the reader needs further explanation the authors recommend consulting any of the manuals of C/C++.

3.1. Declarations

Declarations introduce a new symbol, give its characteristics, type, attributes, and in certain cases the memory allocation of the symbol. Declared symbols point out the objects with certain characteristics. The most important are:

Type - if a set of allowable operations, memory space, and correct bit-pattern is to be determined for an object, the object need to be of a certain type.

Allocation - an object can be allocated in a stack if the local variable is in the static area, that is if it is static or global variable.

Duration - time for which an object legally occupies a particular memory location

Visibility - a part of the text allowing access to the object by giving the name of the object

Category - (**AleC++**) - beside the standard C/C++ categories, object can be a signal, an element. The status of an object will depend upon its category

3.1.1. Basic Data Types

AleC++ has intricate (basic) and composite (derived) types of data. Basic types are:

`void` - absence of type; used for counters and special applications

`char` - character; 1 byte

`int` - integer (UNIX implementation - 4 bytes)

`double` - real type - double precision (UNIX - 8 bytes)

Note**:** Alecsis does not support unlabelled (`unsigned`) operations; type `float` is a synonym for `double`; all integer types are of the same length being 4 bytes. It follows from the previously said that the following types are equivalent:

```
int
short
long
unsigned
short unsigned
unsigned long int
```

Note**:** `unsigned char` is not a legal combination, since all data of `char` type are labelled.

The imposed restrictions are a consequence of the characteristics of the simulation engine that performs the instructions of AleC++ code. Virtual processor is the software emulator of the real microprocessor and it performs those instructions. The need for greater efficiency and speed of the virtual processor eliminated certain types of real (machine) types. These reductions do not limit the capabilities of the language due to the specialized application of AleC++. The reader should keep in mind that according to ANSI-C standard lengths of integer types are interrelated by:

```
short ≤ int ≤ long,
```

which is valid for AleC++, as well.

New symbol can be defined in a way generic to all types:

```
type t;       // variable of type "type"
type *t;      // pointer to a variable of type "type"
type t[2];    // t is a vector of type "type", length 2
type t[2:0]; /* t is an inverted vector of type "type", length 3
              (AleC++ specific !) */
type &t;      // t is a reference (address of data) of "type"
void f(type);     /* f is a function that takes one argument of
                  type "type" and does not return a value */
void (*f)(type); /* f is a pointer to a function that takes one
         argument of type "type" and does not return a value */
```

In general, a simple declaration consists of a type, and declaring phrase, in the following way:

declaring phrase:
identifier
ptr_operator declaring phrase
(declaring phrase)
declaring phrase [constant_expression]
declaring phrase [constant_expression : constant_expression]
declaring phrase (declaration_parameters)

ptr_operator:
** <cv_qualifier>*
&<cv_qualifier>

cv_qualifier:
const
volatile

Qualifiers `const` and `volatile` appear within the declaration of pointers and references, but can appear within a type, as well. In C-jargon, l-value is the expression, which can appear on the left side of the "=" (assignment), that is its value can be changed. All other expressions are called r-value, and they cannot be changed. Qualifier `const` means the object can (and must) be initialized, but that nothing can be stored in it (not an l-value). This is done to protect certain object from change, such as the truth tables of logic operators, various physical constants, etc. You can read more about these qualifiers in the next chapters.

3.1.2. Allocation method

Allocation method is the mechanism by which objects are created during the program execution, or simulation. AleC++ recognizes three ways to do this operations; all of them are marked by key words **auto**, **static**, and **extern**. Due to the characteristics of the virtual processor of the simulator the fourth key word, **register** does not produce any effect, and has the same meaning as **auto**.

3.1.2.1. Auto

Local variables created in the stack, which expands and contracts during the execution stage are marked this way. The duration of these objects is measured from the point of declaration, and ends with the visibility of the object. It is a mistake to process or use the address of such an object beyond the existence range, because they do not exist outside that range. The formal parameters of functions can be defined in this way, only.

3.1.2.2. Static

Static objects are declared at the beginning of the execution phase, and last trough the simulation. In case of a function, the value form the previous call will be used. If a static object is declared on the global level (beyond the function) its existence range is the length of the whole file. In this case the object cannot be externally connected or referred to.

3.1.2.3. Extern

These object are physically situated in some library, and the declaration give the information about the type (but not about the size, since for multidimensional object the first dimension can be left out.) The `extern` declaration makes the object declared in some other file visible in the current file, too.

These objects are classified as static according to the duration; and as global, according to the existence range. During the time of interpreting their final address is not known, which warrants an additional phase before the simulation - linking references of external symbols with the addresses of the objects raised from the library.

Declaring the allocation method is not necessary (**auto** is understood for local objects). In absence of the type **int** is understood.

```
extern char cmatrix[][255];// the first dimension is not necessary
static char *s="static string";
auto i = 2;                    // understood as type int
int i = 2;                     // understood as type "auto"
```

3.1.3. Declaration of new types

AleC++ does not fully recognize new types declaration, since it uses structural and not full type equivalence. According to the structural equivalence, the types are equivalent if they can be separated into the same basic types, while according to the full equivalence those two types would have to have the same type name, without regards to the common basis. To circumvent this obstacle synonyms are created using the key word typedef. A name created using typedef can be used where standard types can.

```
typedef int Meters, Kilometers;
```

We have declared two new names for `int` type. Structurally, those two types are equivalent due to the same basis although the semantics can be incorrect. The syntax of `typedef` is identical as the allocation method description:

```
typedef int  *iptr, ivec[20], imat[20][50];    // new types
static  int  *ptr, vec[20], mat[20][50];       // new static variables
iptr ip;      // ip is pointer of int
ivec iv;      // iv is vector of type int of length 20
imat im;      // im is matrix of type int of length 20x50
```

3.1.4. Address alignment

The address alignment is of importance for installation of Alecsis on different hardware. Great number of microprocessor needs for the addresses of all operands to be divisible by the length of the machine word. On DOS-run computers all addresses need to be even (words are 2 bytes long), while the most of the UNIX-run computers need the addresses to be divisible by 4 (words are 4 bytes long). Certain number of processors (ex. MIPS) needs the addresses to be of double type and divisible by 8. Since the virtual processor of Alecsis works as an emulator of the real microprocessor, it has the same characteristics as the microprocessor of the computer where it is installed.

Alignment method of the computer is easily checked by printing out the addresses of various types and their division by 2, 4 or 8.

Address alignment is of importance only in the installation phase. In Alecsis `Makefile`, flags for alignment are defined. After the installation, address alignment is not of importance for the designer.

3.1.5. Structures

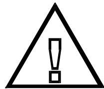
Structures is the common name for three composite types - **`structures`, `unions`, and `classes`**. These represent a collection of heterogeneous data declared without order or number. The total length of structures and classes is equal to the sum of lengths of individual members (or somewhat larger due to the address alignment). Unions have the length equal to the length of the **largest member**. Structures and classes can carry all of their elements simultaneously, while unions can carry only one (since all member of a union begin from the same address). Structures can generally have a name (tag), and may not in case a new type is formed:

```
struct S { int a; double b; char c[30]; };
class C { struct S s; double a[20]; };
typedef union { int i; double d; } INT_OR_DOUBLE;
```

The names of the structures can be used as new types (C++). In C, the name of the structure cannot be used if it is not preceded by the key word `struct/union`. The definition of class K from the above example can be:

```
class K { S s; double a[20]; };
```

If the name of the structure is masked by the name from another visibility area, the key word `struct/union/class` can resolve the ambiguity.



In version 2.0 C++ structures and classes have the same characteristics, except all members of structures have `public` right of access, while classes are `private` (unless defined otherwise). The authors implemented the old C rules, reading that a **function cannot be a member of a structure**. If it makes you a problem, instead of a structure, you can use a class with all members explicitly set to `public`.

A definition of a structure can simultaneously be a declaration if variables are added:

```
struct S { int a, b; } s1, s2;    /* defines structure S and
                                     variables s1 and s2 of type S */
struct { int a, b; } s3;  // a structure without a name
struct { int a, b; };     /* a futile definition,
                          interpreter reports an error */
```

Bit-ranges are a separate structural type. Their meaning is fundamentally different from ordinary members, although they are declared similarly:

```
struct flags {
    int f1:2;
```

```
        int f2:3;
        int f3:1;
    };
```

Member `f1` occupies 2 bits, `f2` - 3 bits, and `f3` - 1 bit. The total length of the structure is 6 bits, but due to the address alignment at least one word (4 bytes) will be allocated. Access to the members is the same as with the ordinary structure, but the user need to take care that the assigned values do not overflow the range defined by the number of bits following the character.

3.1.6. Enumeration type

Enumeration constants were mentioned in the previous chapter. To remind you, in AleC++ enumeration constants can be used only in the context of the variables of the same type. This implies that by the definition of the enumeration set a new type is declared:

```
enum bit { '0', '1'};                    // enumueration with name
typedef enum { '0', '1', 'x', 'z' } four_t; // enum. as a new type
enum { send, recv };      // futile declaration - error
bit b[4]="0011";          // vector of type bit
four_t s = 'x';           // scalar of type four_t
```

3.1.7. Multidimensional variables

Standard variables can be considered scalars, because a unique memory location exists for every one, which can be accessed using the variable name. Vector variables reserve memory space for as many members as indicated in the vector dimensions.

```
int i[10];   // vector of type int of length 10 from i[0] to i[9]
double m[5][10]; // matrix of type double of length 5x10
```

As in C, all multidimensional variables have 0 offset (vector *v*, of length *n* reserves locations *v*[0]-*v*[*n*-1]. Beside that, new allocation methods are introduced in AleC++, since all vectors are used for modelling of abstract registers (signal vectors and node vectors). If the length of the vector is defined using upper and lower limit the vector can have a nonzero offset. In case the upper limit is smaller than the lower we are talking about inverse direction:

```
int i1[10];        // vector of length 10
int i2[0:9];       // equivalent declaration, but with limits
int i3[1:10];      // vector of length 10, with the offset of one
int i4[9:0];       // vector of length 10 with inverse direction
int i5[10:1];      // same, w/ the offset of one
```

Note: it is incorrect to access indices outside their declared range. Location `i3[0]`, for example does not belong to the vector `i3` the same way the location `i3[11]` does not. Vectors with the inverse direction behave the same as the direct vectors except the order of assigning the values is reversed during the initialization.

```
digital3 v1[0:3] = "010x";      // position v1[0] has value '0'
digital3 v2[3:0] = "010x";      // position v2[0] has value 'x'
```

If the name of a multidimensional variable is found in the expressions it is implicitly converted from "vector v of type t" to "pointer of location [0] of vector v of type t." If the lower limit is not 0, the pointer will be point to

the first element marked by the limit (`[1]` in vector `i3`). Inverse direction vectors will have the pointer show the element with the lower index (`[0]` for vector `i4`).

3.1.8. Initialization

The rules of initialization in AleC++ are slightly different compared to C++, in order for the inverse direction vectors to be correctly initialized. Object with the automatic allocation can be initialized using user-defined expression, but only if the expression evaluates to a scalar. Static or global objects can be initialized using constant expressions only; initialization of composite objects (vectors, matrices, strictures, etc.) is legal, as well.

```
int i = 2;        // automatic initialization
int j = i+1;      // this too
static char *s = "constant string";    // O.K. - static object
static char *s[] = {"first", "second", "third", "end"};//composite
double m1[][3] = { { 3,  3, 2},
                   { 2,  2, 1},
                   { 0, -1, 2} }; /* every char '{' reduces the
                                      dimension by 1 */
double m2[][3] = { 3,  3,  2,
                   2,  2,  1,
                   0, -1,  2 }; /* O.K. - interpreter can find
                                      its way around this */
struct S { int x; double d; char *s; };
S s1 = { 1, 2.2, "string" };
S s2[2] = { { 1, -2., "s1"}, { 4, 5.6, "s2" } };
```

As you can see the interpreter can calculate the first dimension of a vector based on the initializing phrase. Composite object is initialized from left to right according to the order of specific values in the initializing phrase, with the exception of the inverse direction vectors, which are initialized from right to left. Alignment of object types and initializing phrase is controlled during the initialization according to the standard assignment standards (=) with the implicit conversion, if necessary.

Static objects are initialized once, at the beginning of the program work cycle. Dynamic (automatic) objects are initialized every time the flow of the program comes across an initialization, while the present initialization ceases to exist with the object itself.

3.1.9. Declaration of functions

Functions occupy the central place in AleC++, which supports all rules of declaring and using functions in C/C++.

Note: Before the function call you need to define the function declaration (consist of the type the function returns assigned to its name, and a string of parameter, if any). Function that returns no value is labelled `void`**.** Parameter of the function that does not accept any parameters is also labelled as void.

Note: There is a fundamental difference in the way of declaring functions in old and new ANSI-C:

```
int f1 ();
int f2 (void);
double **f3(char *, double)
char *f4 (int (*f)(int, double));
```

According to the old C (Kernighan-Ritchie) function `f1` returns `int` and accepts unlimited number of parameters. In ANSI-C, C++, and AleC++ the function returns `int`**,** but does not accept any parameter (identical with the declaration of `f2`**).** Function `f3` returns the pointer to pointer to `double` type, and accepts two parameters: one pointer to `char` type, and another of `double` type. The last function, `f4`, returns the pointer to `char`, and accepts one parameter that is the pointer to the function that returns `int`, and accepts two parameters - of `int` and `double` types.

In the declaration of the prototypes of functions the names of the functions are not important, since the prototypes (parametric profiles) of functions are used for check of the legality of the function call (regarding the number and types of the actual arguments compared to the expected ones). We use prototypes of functions in case of overload (many functions with the same name, but different parametric profile.) To summarize, we use function declaration:

- to check the agreement between the number of expected (formal) parameters, and the number and types of the actual arguments at the time of the function call;
- for application of standard conversion types according to the mechanism used in initialization or assignment (=);
- to reach a decision about the nature of the overload function (overload will be discussed in detail in the next chapter.)

3.2. Definitions

An object is declared for the purpose of giving information about its type, dimensionality and other aspects needed for its processing. In case declaration gives information about the memory reserved for the object, as well as the information about the internal structure, it is a **definition**. If on the global level, declarations:

```
extern int vec[2][3];
extern char s[];
```

are declarations only, for they are used in order for the interpreter to create a correct code of expressions in which matrix `vec`, and string `s` are used. Declarations

```
int vec[2][3];
char s[20];
```

are at the same time definitions, since they give the information to the interpreter to reserve 2x3x4=24 byte of static memory for the matrix `vec` and is 20x1=20 bytes of memory for string `s`. **There can be many declarations of the same object** within the same file (and all the other files added by `#include` preprocessor directive), **but only one definition.** Global (or static) data or functions can be defined on the global level. AleC++ introduces two more fundamental objects that can be defined - **module** and **module card**, but you can read about that in the chapter about the simulation.

3.2.1. Definition of functions

All functions are defined on the global level. Definition of functions is similar to their declaration, with two extra conditions:

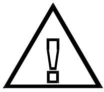
- The names of formal parameters need to be specified (if the parameter are needed), beside the types.

- It is necessary to define the body of the function in the extension of the prototype consisting of the array of declarations and commands enclosed by parentheses '{' and '}'.

```
main () {
    printf ("Hello, object-oriented world of simulation!\n");
}
```

In C, functions can be called by other functions according to the program algorithm. The program begins from the **main** function. In AleC++, functions are usually called from concurrent processes during the simulation; however, it is legal to write a program that has only C or C++ constructs, and the function `main`. If no module labelled **root** exists (a simulation counterpart of `main` function) in the originating file the simulator will deliver the address of the function to the virtual processor to commence execution.

Therefore, AleC++ can be used as an interpreter for C/C++. It is useful for development of new functions. A part of some complex component model can be developed as a library function. Such functions can be tested in the same way as in C before the simulation constructs are added. More reliable, and more error-free models are created this way.



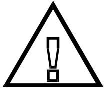
AleC++ is designed as a superset of C/C++. However, some constructs of C++ are not allowed in the current version as we were oriented mostly to construct that we need for simulation. **Not implemented** are:
- virtual functions,
- virtual base classes,
- conversion functions,
- copy constructor,
- implicit conversion using constructors,
- `new` and `delete` for vectors.

All variables in a function are allocated in the stack (including the formal parameters), except the ones explicitly defined as static. Formal parameters create special visibility area, which is narrower then the global level. By creating a new visibility area inside a bigger one, in general the possibility exists for the variable under the same name to be declared without the danger of redefining it. In this case the variable in the wider area masks the one in the narrower, until the narrower area is left.

```
int i;                // global i is visible
f1 (int i, int j) {  // formal i masks global i
    int j;            // local j masks formal j in the whole fn
    if (i>>2) {
        int i=3;      // formal i is masked, as well
        ...
    }
    ...               // formal i is visible again
}       // the body of the fn is closed
// the area of the formal parameters is closed
// global i is valid again
```

Actual arguments (expressions) are stored in the stack during the function call, according to order of specification, and are passed by **value.** Passing the argument value to functions (and not the argument address) is a characteristic of C, and AleC++ supports this mechanism. It is possible to pass an argument according to the address if the formal parameter is the argument reference (e.g. `&a`). You can read more about that in the chapter on object programming.

Passing by value means the functions gets a copy, and not the original of the parameter, so the change in the copy does not affect the original. **All objects** are passed by value including structures or large classes, except multidimensional variables, which are converted into pointers to the first element. This means that the original and not a copy of the pointer is passed to the function, which implies that matrices, and vectors can be changed inside a functions. In order to avoid any unwarranted change in these objects, the formal parameter can be labelled **`const`**, which prevents accidental change of a vector or a matrix.



AleC++ does not support the old way function parameters declaration during their definition. This means that definition

```
int f1 (a, b) double a, b; { ... }
```

is not correct. Function f1 has to be defined in the following manner:

```
int f1 (double a, double b) { ... }
```

3.3. Expressions

All legal combinations of identifiers and operators, which follow syntax rules, are expressions. Among these are rules on right-a-way of the operator, alignment of types, etc. Usually, these expressions are combinations of symbols and arithmetic-logic operators.

Note: The result of the application of the operator on the expressions is also an expression.

AleC++ supports all rules for expressions of C/C++, and introduces some new ones, which will be discussed in detail in the chapter on simulation.

An expression is constant if it is entirely made out of combinations of constants. A vast number of situations in AleC++ demand a constant expression. Alecsis evaluates the constant expression during the compilation. Therefore, the expression

```
50 * 2 / 5      // can be evaluated
```

does not cause creation of instructions for multiplication and division for the interpreter, since the compiler evaluates it as an integer constant 20. In a similar way, implicit conversion, in case of combination of a variable, and a constant can be accomplished without the creation of instruction by a direct conversion of the constant. Example is:

```
double d;
d+1;            // 1 is converted into 1.0
```

An expression is unary if it consists of operators and expressions, and binary if the operator needs two operands, which are also expressions. All expressions in AleC++ are unary or binary, with the exception of conditional expressions (as in C/C++), which demands three expressions:

conditional_expression:
expression1 ?expression2 : expression3

The number of possible combinations in expressions is very limited, which is why the interpreter in dealing with binary expressions with different types applies the standard rules of conversion or **promotion,** in order to have both operands as of the same type. Promotion in arithmetic expressions means that the expression of a 'lower' type is converted to the 'higher' type, which becomes the type of the result. In the case of assignment, the rules demand for the right-side expression to be converted to the type of the left-side expression before the operation is performed. The rules of promotion and conversion of types are applied in the following cases:

- in all binary expressions;
- during the initialization;
- before the actual arguments are passed to the function;
- when the result of the function is returned using the command `return.`

In all these cases, except the first one, the rules of conversion of the assignment operation (=) are applied.

```
2 + 3            // constant expression
a ( 2, 3 )       // function call with two arguments
b[4]             // indexing of a vector
s->>m.a          // selecting of structure members
a >> b           // relation expression
++a, a++         // left and right increment
a += 2           // appended addition and assignment
a,b,c            // coma operations(developed from left to right)
a==b ? c: c+1    // conditional expression
a + (b + c)      // grouping using parentheses
```

3.4. Commands

The program is a set of declarations and commands. The meaning of a command is to change the state of the system by its execution. If that is not the case it is a null-effect command. The simplest command of AleC++ is an expression finished by a semicolon “;”.

```
i+2;
```

This command is legal, but has no effect, since nothing happened after the execution. Most of the commands include an assignment operation or a function call. Commands can be simple or composite. Composite commands are a set of commands and/or declarations enclosed by parentheses {}. This **block** creates a new name space. This allows for the masking of the variables with the same name in the external space.

All commands are executed according to the order of specification. The flow of the execution can be controlled using branching commands, **`if/else`.** A sequence of numbers can be repeated by using loops - **`while`**, **`do`**, and **`for`**. A loop can be interrupted using command **`break`**. A present loop cycle can be interrupted, using command **`continue`** (new cycle begins after that). Functions can return expressions to the environment they have been called from using **`return`**. If testing the value of a scalar integer expression using multiple options is needed, the command **`switch`** can be used.

Identifier **label** commands by putting the label and colon in front of the command. The purpose of the label is to mark the place for unconditional jump using **`goto`**. Note: The use of `goto` is recommended only in the extreme cases where the speed is more important the legibility of the code.

Two characteristics of C++ are adopted, that do not exist in C. First, commands and declarations can freely intermix within a block (C demands that declarations are placed at the beginning of the block, before commands). Second, the first of the three expressions in the `for` loop specification can be a declaration. For example:

```
{
  int i=1, j=2, k;          // one declaration
  k = i + j;                // command
  int l = k+1;              // declaration, again
  for (i=0; i<10; i++);     // empty for loop in C-style
  for (int q=0; q<10; q++);  /* q is in the same visibility area
                                 as the other variables */
  double d = k-j;           // another declaration
}
```

New variable, introduced during the initialization of the `for` loop in the example has the same existence range as the others, i.e. it is visible in the block where `for` loop is.

Command **`asm`** is unique to AleC++. It enables direct insertion of the assembler instruction in the C++ -like code. The assembler code for virtual processor is prepared by the compiler, but it can be done directly by the user, using assembler instructions. The set of instructions resembles the instructions for Motorola processor series 68020/30, and it is given in the Appendix. Programmers of low level libraries, and functions used frequently in modelling will find this to be effective shortcut to writing fast programs by eliminating unnecessary instructions.

This is the syntax for **`asm`**:

assembler:
    asm *asm_text ;*
    asm *{ sequence_asm_lines }*

sequence_asm_lines:
    asm_text
    sequence_asm_lines asm_separator asm_text

asm_text:
    opkod<.optip> <operand1> <, operand2>

asm_separator:
    '\n'
    ';'

This an example of the command:

```
void f1 (int i, int j) {
    int l=3, m;
    asm {
        add.l        i, j
        movq.l   %d4, %d0
        add.l        l, 2
        sub.l        %d4, %d0
        movq.l   m, %d0
    }
}
```

In the previous command variables `i` and `j` were added together, first. Suffix l marks the 'long' version of the instruction (suffix `b` means 'byte', and `d` 'double'). The result of addition is placed in the accumulator (`%d0`). It is temporarily moved to `%d4` registrar (instruction `movq`) to free the accumulator for other operations. Local

variable `l` is added together with constant `2`, and the result is subtracted from the value in the register `%d4` (`sub`). Finally, the result is assigned to the local variable `m`. As you can see from this example, even simpler expressions, names of variables, etc. can be operands of instructions.

You can translate AleC++ code into corresponding assembly language code The compiler create it by using option **`-S`** from the command line. The file created in this way has extension `as`.

4. Object-oriented programming

The given overview of lexical and grammatical rules focused on the common points between C/C++ and AleC++. This chapter will also focus on similarities between AleC++ and its basis languages. If the reader is not familiar with the object programming introduced by C/C++, and this chapter is too succinct to explain it completely, the authors suggest consulting one of the numerous manuals on the subject.

Object-orientation enables better reusability of already developed program parts. With this concept software is developed in phases, layer-by-layer. Diverse, multifarious building blocks are developed first, and then used to put together to form larger blocks, which can be used again to build some other blocks. Alecsis was developed to model large number of various electronic components and systems. Many of them can be considered as composed of some building blocks. That is why the authors included the construct of object programming in AleC, and turned it into AleC++.

Before we approach object programming we will discuss some rules of C++, not necessarily directly connected to object programming, since we will need them in the coming chapters. Those rules address functions overload, standardized constants, inline functions, initial values of formal parameters, etc.

4.1. Constant variables

You already read about constants in the chapter on lexical rules. It is possible to name a constant by using the preprocessor command `define`, and then use that name in the program.

```
#define PI 3.141
```

ANSI-C and C++ introduce variables whose value should not be changed in the program.

```
const double PI = 3.141;         // constant of type double
const No = 123;                  // constant of type int
```

These constant has to be initialized immediately upon declaration (unless it is an external variable), since that is the only place initialization can be done.

Interesting effects can be produced using combinations of qualifier `const` and various pointers, as the following example shows:

```
const c=2;              // constant, int type
const *pc = &c;         // pointer to a constant
int *const cp = pc;     // constant pointer
const *const cpc = &c; // constant pointer to a constant
int *vp = &c;   // error - pointer must not point to a constant
```

Pointer `pc` can be changed afterwards in the program, but not the value stored on the address it points to. This is possible to do in the case of the pointer `cp`, but in this case the value of the pointer cannot be changed. Finally, neither the value of the pointer `cpc`, nor the value on the address it points to can be changed. The last example illustrates a semantic error -- if it was allowed for the pointer `vp` to be set to the address of constant `c`, the value of the constant can be changed indirectly using the pointer.

Qualifier `const` can be used with function declaration, too. The declaration of standard library function `strcpy`, given below, indicates that only string `dst` changes within the declaration block. Using this technique, an error can be detected if a pointer to a constant is passed to a function where it can be changed.

```
char* strcpy (char* dst, const char* src);
```

4.2. References

All arguments are passed to functions by value, which means local copies of original arguments are created, and a change in a copy does not affect the original. The pointer to the variable needs to be passed to the function, if the programmers want to change the variable inside the function. A certain number of situations exist where we want the same effect without the use of operator `&`. We refer to passing of large object by reference in order to save time. It is possible to declare object in AleC++ as a reference to another object of known type. We will encounter this situation when we declare *formal parameters*, *local object*, and *the type of the data that the function returns.*

4.2.1. Formal references

If you declare a formal parameter of a function as a reference, the address of the argument will be passed to the function and not the copy of the argument. Then, every change of the parameter changes the argument. Declaration of a reference is the same as in the case of the pointer, except we use the operator `&` instead of `*`:

```
f1 (int pi)  { pi++; }
f2 (int& pi) { pi++; }
f3 () {
    int i=1;
    f1 (i);    // after return, i is still 1
    f2 (i);    // after return, i is 2
    f2 (i);    // i is now 3
}
```

If you do not like the idea of being able to change an argument in a function, you can always declare the argument as `const`. You cannot change the argument now in the function, but using references, you can still save some time in the transfer of large objects.

4.2.2. Local references

Reference object inside a function is practically another name for the initializing object. **Local reference has to be initialized immediately after declaration**, bearing in mind that every change in the reference will affect the original object.

```
int i; int& ri = i;    // ri is a synonym for i
ri = 2;                // i is now 2, as well
```

4.2.3. Reference-returning functions

A function can return a reference under its name. In this case the function call can be an *l-value*, meaning that you can write into the function call. This property is used mostly with the overloading of some operators. **The function has to return a valid reference, which is not a local object**, because local object is erased during the return from the function (dangling reference).

```
int &f1(int, int);
...
f1 (2,3) = 4;
```

In all three cases given above, if the initializing object is not an *l-value*, or if the rules demand type conversion, a temporary object will be created, and its address will be used. As this is usually unwanted, the compiler will warn you.

4.3. Function overload

Multiple functions of the same name can exist in C++ and AleC++. The compiler will not report redeclaration if all of them have different parametric profile (type, and number of parameters). In the moment of the function call, compiler decides which function satisfies the actual call the best, taking into consideration the number of implicit conversions that will have to be applied to arguments.

```
double f1 (int);      // the first declaration
double f1 (int);      // copy - O.K.
int f1 (int);         // error - difference is in the returned type
double f1 (int, const char*); // O.K. - different profile
...
double d1 = f1 (2);           // call to f1(int)
double d2 = f1(3, "string");       // call to f1(int, const char*)
```

Compiler will easily find its way around an overload of the name of the function, if it has the information about the profile. The linker, however, cannot determine the version based solely on the name of the symbol.

Alecsis linker uses **type-safe linkage**. It secures the uniqueness of the function name by adding an especially coded parameteric profile (name mangling) to that name. For instance, the first version of `f1` in the symbol-table of linker will be `f1_i` (having one integer parameter), the second `f1_iPc` (having one integer and one pointer to character as parameters).

One should note that the function overloading is very useful for simulation, especially logic simulation. One can create a single name for a logic operator (for instance, `and` gate) which can have two or three inputs.

4.4. Default values of function parameters

We expanded the ability to have default value of parameters in Alecsis to include specific simulation constructs. If an actual argument does not appear on the appropriate place, its default value is placed there. This enables function calls in many different ways, with the parameters always defined.

```
f1 (int i, double j, int k=2);  // k has the default value 2
int p = f1 (2,2.3);             // parameter k is set to 2
int q = f1 (2, 5.1, 7);         // here, k equals 7
```

The only restriction in the number and type of the parameters with initial values is that **all parameters to the right from the first parameter with the initial value have to have initial values**, as well. An overload of this function can confuse the compiler:

```
char *f (int, int=2, int);     // error - the third argument does
                           // not have a default value
int *fp (int), *fp(int, double=2.2);   // two versions of fp
int *p1 = fp(1, 5.6); // O.K. - call of the second version of fp
int *p2 = fp(1);  // ??? - both variants are suitable - conflict
```

Note: All initial values have to be constant expressions.

4.5. Inline functions

Ordinary functions are translated and stored in the library, and linker finds them if used in the code. It pays off to insert the function in the code instead a function call in case when the function is only a few lines long, as the time needed to for the function call is longer that the time of function execution in such case. The **precondition is that the function is already defined in the same file** using the key word `inline` before its definition. This is used to speed up significantly the execution of the program. However, certain restrictions apply, such as: inline functions cannot have any loops or unconditional jump commands, cannot declare static variables (local yes), cannot pass their address, cannot be recursive, etc. They are most useful when working with small, but often called functions.

```
inline double sum_a_b(double a, double b) { return a + b; }
```

4.6. Functions with variable number of arguments

This feature is a part of C, and it enables the writing of functions like `printf`.

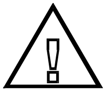
```
#include <varargs.h>
void print_menu (const char *title, ...) {
    char *field;
    char *args;

    printf(menu %s options:\n", title):

    va_start(args, title);

    int nmenu=0;
    while ((field=va_arg(args, char*))!=0) {
        printf("\t%d: %s\n", ++nmenu, field);
    }
}
```

At least one argument must be given in the list of formal arguments. The last given parameter (in our example, there is only one given parameter -- `title`) is used as the argument for macro `va_start` from `varags.h` file. Macro `va_start` sets pointer `args` to the memory location **after** the parameter title, i.e. to the memory where the next argument is. All other parameters will result from the call of macro `va_arg` by passing the initialized pointer `args`, and the expected parameter type. The type has to agree with the type of the argument, since the compiler is not able to perform implicit conversions in this case.

Note: If the parameter to be read using `va_arg` is of type `char` or of some other short type, the second parameter of `va_arg` must be `int`.

Note: Coma is not necessary in between the last defined parameter and the symbol '...':

```
void print_menu (const char *title ...);
```



Implementation of functions with the variable number of arguments utilizes the fact that all formal parameters of the function are stored in consecutive memory locations. However, some computers have special alignment rules. Computers *Silicon Graphics* and *HP 9000 s700/800* store parameters of type `double` on the memory locations that can be divided by 8. This option is implemented in `varargs.h` file, you just have to define `DWORD_ALIGNMENT` flag before including this file:

```
#define DWORD_ALIGNMENT
#include <varargs.h>
```

For other computers where Alecsis until now installed, such alignment is not necessary. If you install Alecsis on some new type of computer, that is not predefined in `Makefile`, you should see the alignment rules for that computer in file:

```
/usr/include/varargs.h
```

that is used by C. You can then adapt Alecsis `varargs.h` to fit your needs.

4.7. Visibility area resolution operator

This operator is the first in the series of operators not used in C. This operator (two consecutive colons (`::`)) was designed to solve the situations when it is not clear which variant of an object is used, in case when symbols with the same name exists in different visibility areas. It is widely used as a part of objects whose type is a class, but it can be used independently:

```
int i;              // global i
main () {
    int i, j;       //local i and j

    j = i;          // refers to local i
    j = ::i;        // refers to global i
}
```

4.8. Classes

Classes are a special case of composite types, similar to structures. They allow the expansion of the existing system of types. Classes can have members -- data, but unlike structures, classes can encompass a number of appended functions. These appended functions are methods, which manipulate the objects, overload existing operators to make them applicable to newly defined types, as well as provide mechanisms of implicit conversion in case of dealing with "changed" types. Fully defined class represents a type, equal in status to already existing types.

We emphasize that classes differ form structures in that classes can have functions for members (in version 2.0 C++ structures can too, but this was left out in AleC++). Access to members stays the same as in C - operators '`.`' for objects, and '`->`' for pointers to objects.

```
class X {
    int a; double b;
    char s[20];
};

main () {
    X x, *px;

    x.a; x.b;
    px->a; px->b;
}
```

4.8.1. Access to class members

Members of classes are data and functions. They are **local** in reference to the class, and cannot be referenced outside the function. These functions are methods, since they define the operations with the objects of the class. The access to the members of a class is not simple doe to the need to control the access. Members of a class are initialized as `private` (that is no one except their own methods have the right to access them). Members and methods have to be defined as `public` to be accessible. We will talk about the third type of access -- `protected` in the chapter on inheritance.

4.8.2. Declaration and definition of methods

Methods declared in a class have to be defined somewhere. Shorter methods can be defined within the class, which makes them `inline` functions. It is possible to use members of the class not yet declared (as in the case of definition outside of class) within the methods defined within the class, since those methods are not translated until the class is entirely defined. Methods defined outside a class can be `inline`, too, but the key word has to be used in this case.

The following is an example of a class:

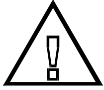
```
class Point {
        int xVal, yVal;
   public:
     void set () { xVal = yVal = 0; }
     void set (int x, int y) { xVal = x; yVal = y; }
     void show() { printf("xval = %d, yval = %d\n", xVal, yVal); }
     int Xval () { return xVal; }
     int Yval () { return yVal; }
};
```

The class `Point` has only two data of `int` type - `xVal` and `yVal`. These are private, and cannot be accessed from outside, since only class methods have the right of access. All methods are defined within the class, and are therefore inline. Overloaded method `set` gives them their values, `show` displays them on the screen, while methods `Xval` and `Yval` return their values. These are the only legal operations with the object of this class. Restriction of possible operations on members of a class is a valuable advantage of object programming, since it limits who and how can change objects of a particular class. Errors are easier to find, since it is always specified who is responsible for a particular operation.

```
f1 () {
    Point p1, p2;
    int x,y;

    p1.set();              // p1.xVal and p1.Yval are set to 0
    p2.set(2,3);           // p2.xVal = 2, p2.yVal = 3
    p1.show();
    p2.show();
    x = p1.xVal;           // error, xVal is a private member
    y = p1.Yval();         // O.K. - Yval is a public method
}
```

4.8.3. Keyword this

In the previous paragraph methods have referenced members of the class, as if these were known to them in advance. All methods have an additional level (besides usual levels of visibility) -- class level. This level is narrower in the order of searching than the global, but wider than the level of formal parameters. This means that a class member masks global variable of the same name, while a formal parameter or a local variable masks the class member. **The class object, whose members are used is passed as a hidden parameter in every appended function.** This can be made visible using the keyword `this`. This keyword represents exactly the passed object, and is manipulated as any other object of the same type.



In C, the keyword `this` refers to the **pointer of the object** passed to the appended functions, so the access to the individual members requires the operator of indirection `->` (eg. `this->xVal`, `this->yVal`). Alecsis virtual processor uses a table of pointers to the sources of the most frequent operands, and is able to **treat `this` as an object**, and not an operand. This renders dereferencing of the word `this` unnecessary, which can save time in larger methods. Access to particular members is possible using operator "`.`" (eg. `this.xVal`, `this.yVal`).
Nevertheless, we intend to change this to be the same as in C in the following releases of Alecsis, not the confuse the user.

You can explicitly use `this` in order to differentiate the names of the class members from names of other variables, in case of masking of the members of the class. The second option is to list names of classes and using the resolution operator:

```
class X {
        int x, y;
    public:
        void set1 (int x, int y) { this.x = x; this.y = y; }
        void set2 (int x, int y) { X::x = x; X::y = y; }
};
```

4.8.4. Static methods and class members

It is not legal to use all specifications the method of allocation (extern, register, auto) inside the body of a class. Mentioned specification can be applied to the members and functions, with different affects.

4.8.4.1. Static members

When objects of a class are defined, every object generates its own copy of the individual members, which occupy different memory locations. If we want the present state of object characteristics, including number, state, etc., static members can provide the necessary information. **There exists only one copy of the static member of the class, no matter how many copies class objects exist**. These members are internally implemented as static, or global variables, and do not affect the size of the class (only non-static members do). Declaration of the static element is not a definition, therefore in the case of the class on the global level it is the same as for any other global variable, but using access operator:

```
class X {
    int a, b;
    static s;    // declaration of the static member of the class
                 // of int type
};               // class occupies 8 bytes

int X::s = 0;    // definition of the static member of class X
```

Access is the same for static members as for any other member of the class:

```
X::s = 3;        // access beyond any object
X x;             // object x class X
int xs = x.s;    // access using object x
```

4.8.4.2. Static methods

These functions are defined as methods, but are very similar in behaviour to global functions. These functions do not accept current class object as a hidden parameter, since they are used outside an object, so key word `this` cannot be used with these functions. They have to accept a pointer to an object explicitly in order to use it:

```
class X {
        int xval;
    public:
        static void fx (class X*);
};
...
X x;                  // object x

X::fx (&x);//call of static function independently from the object
x.fx (&x);            // classical  method of access

void X::fx (X *x) { // definition of static fn. fx, class X
    xval = 1;         // error - which xval is this referring to?
    this.xval = 1;    // error, as well
    x->xval = 2;      // O.K.
}
```

Static member and functions lessen the congestion of the global area with symbols, help define which global or static symbols belong to which class, and offer the ability to limit the access using `private` keyword.

4.8.5. Class friends

One may need that other functions, not only the class methods, are allowed access the private members of the class. To enable that, one may declare functions or classes as *friends*.

4.8.5.1. Friendly functions

A function can be defined as a “friend” of the class by using the key word `friend`. A friendly function has the right of access to private members of the class:

```
class Y;                    // incomplete Y class declaration
class X {
        int xval, yval;
    public:
        set (int x, int y) { xval = x; yval = y; }
        friend int f1 (X *);  // global function f1 is a friend
```

```
        friend Y::fy (X *);    // method fy of class Y is a friend
};

class Y {                   // ending of class Y declaration
        int Yy;
    public:
        fy (X*);
};

int f1 (X* x) { return x->xval + x->yval; }     // definition f1
Y::fy (X* x) { return x->xval + x->yval; } // definition Y::fy
```

Friendship declaration does not mean passing an object as an implicit argument, thus an object must be passed explicitly to the function. Therefore, global function cannot use `this` even when declared as a friend. The example above shows that a method of one class can be friend of another. The friendship declaration does not change the meaning no matter what area of the class it is situated in (public or private). A method and a friendly class with the same parametric profile do not cause warning messages about redeclaration.

4.8.5.2. Friendly classes

If the intention is for all the methods of one class to have access to all the members of another class, the first class can be declared as friendly:

```
class X;
class Y {
        int xval, yval;
        ...
        friend class X;        // can do without "class"
};
```

This approach makes the job easy, but you need to be careful, since if everyone is defined as friend than the whole system loses meaning since everyone can change data.

4.9. Constructors and destructors

It is possible to declare arbitrary number of methods inside a class. Those methods are called explicitly, as any other function. Some other functions have special meaning, and are the basis for object-oriented programming in AleC++.

Variables declared within an area of visibility (most common way - using '`{`') have existence period ending with the closing of the area of visibility (most likely -- using '`}`'). Compiler allocates enough space for those variables, and consequently frees the space after the area of visibility is closed. In order to allow the class object to behave as object of predefined types (`int`, `double`, etc.) we need to define constructors and destructors as separate methods.

The purpose of the constructor is to allocate memory space for pointers, which are members of classes, and to initialize them whenever an object of that class is declared. Destructor does the reverse -- it frees the memory occupied by a constructor (deinitialization is not necessary). Destructor is not necessary for class objects without pointers, because there is nothing to free, but the constructor is.

4.9.1. Constructors

Constructor is the method that has the same name as the class. Constructor does not return anything, but it can have parameters. It can be overloaded. Constructor needs not to be called explicitly, since compiler calls it whenever it creates an object. In order to make that happen, constructor has to be defined as `public`.

```
class Point {
     int xVal, yVal;
  public:
     Point (int x, int y) { xVal = x; yVal = y; }
     Point () { xVal = yVal = 0; }
     void show () { printf("xval= %d, yval = %d\n", xVal, yVal); }
};
```

Arguments are defined in case the constructor has parameters:

```
Point p1, p2(2,3), p3 = p2;
```

Object `p1` does not have arguments, so compiler calls constructor without parameters. Object `p2` uses constructor with two `int` type parameters, while `p3` is initialized by copying the object `p2` (constructor is not invoked). If one of the mentioned ways is not used, compiler will report an error.

In the case of object `p3`, compiler made a shallow copy of the object `p2` by copying its bit-pattern. If an object involves a pointer pointing at allocated memory, it is necessary to define a special type of constructor -- copy constructor, which accepts reference to a class, e.g. `Point (Point&)`. The object `p2` would be passed by reference into that constructor, where space would be allocated for the memory pointed by the pointer. In that way, the deep copy of the object would be made.

Constructor can be called in expressions as an ordinary function. Than it creates temporary, nameless object, which is destroyed after the exit from the visibility area. Initialization can be done using this temporary object.

```
Point p1(2,3), p2 = Point(4,5);
```

This means going the long way, but temporary objects play an important role in the in operator overloads and implicit conversions.

Constructors for the objects created on the stack are called every time the control gets to their declaration place. Constructors for global, or static variables are called before the beginning of the simulation (if one uses Alecsis simulator), that is before the `main` function (if one uses AleC++ for C++-like programs). Constructors cannot be static functions. They can have parameters with initial values, just as any other function.

4.9.2. Destructors

Destructors bear the same name preceded with the character '~' as the originating class. Destructor do not return any result, nor accept any parameters, which is the reason why destructors cannot be overloaded. Compiler calls them immediately before an object is destroyed in order to free the memory occupied by the object, except in the case of global, or static variable when they are used at the and of the simulation (or on leaving the `main` function). If the command return appears before the end of a block, destructors are called before the exit from a function. As in the case of constructors, destructors cannot be static objects.

```
class X {
        ...           // class members
    public:
```

```
        ~X();         // Destructor for class X
};
```

4.10. Operator overload

C programmers know it is possible to declare variable types of structures, but that little can be done with them using current C operators. Structures can be copied into each other, can be passed to functions, returned using `return`, and that is it. (We are describing operations with objects, and not their particular members.) Structures cannot be added because compiler does not know how to do it, that is instructions for addition of operands that are not integer or real do not exist. Operator overload mechanism "explains" to compiler how to apply the existing operators to object of class type.

4.10.1. Global level overload

Overload of operators is possible by defining identically named functions to be called when needed. Word `operator` should be put before the operator name, which is the special signal for compiler to transform it into a function name.

```
class complex {
       double re, im; // real and imaginary part of complex number
    public:
       complex (double r=0.0, double i=0.0) { re = r; im = i; }
       friend complex operator+ (complex, complex);
};
...

inline complex operator+ (complex l, complex r) {
    return complex (l.re+r.re, l.im+r.im);
}
...
foo () {
    complex c1 (2,3), c2 (7,8), c3 = c1 + c2;
}
```

Class `complex` from this example is a new type of data that represents complex numbers. The class constructor defines the number, and becomes a complex zero in the case of left out arguments. Usually, compiler would not know to add two complex numbers, and would report an error. However, operator '+' overloads on the global level if used on two objects of the same type. Since the operator function is global, it could not access to private members unless previously defined as friendly within the class. The function is defined as inline**,** due to its shorth length. Object passes to it using shallow copying (by value), which is satisfactory in this case, since they are only 16 bytes long, without pointers. The temporary object created in the operator function returns to the environment using command `return` (shallow copying). Here, it initializes object `c3`, which gets the value of 9+11i.

All operators in AleC++ can be overloaded, except:

```
.   .*   ::   ?:   <-   now   $   $$   @   ddt   d2dt2   idt   sdt
```

In general, binary operator ♦ in the expression `op1♦op2` can be overloaded using global function `operator♦(top1,top2)`, where `top1`, and `top2` are appropriate types of operands with the allowed

implicit conversions. Unary operator ◊ in the expression ◊ op can be overloaded using global function `operator◊(top)`, where `top` is the type of operand with the allowed implicit conversions. If the objects are large, it pays off to pass them to the function by address (binary operator function would be declared as `operator♦(top1&,top2&))`.

Operator function can be called as an ordinary function, i.e. as affix:

```
c3 = operator+(c1,c2);
```

but it is a matter of style to use it as an infix operator.

4.10.2. Overload using methods

An operator function can be a method of the class, in which case all rules for methods apply.

```
class complex {
    ...         // the same as in the class "complex" given before
    complex operator+ (complex r)
        { return complex (re + r.re, im + r.im); }
};
```

The method `operator+` is implicitly defined as *inline*. In binary operations, those methods have **one parameter only**, since the left operand is implicitly passed (`this`). Unary operator methods **do not have parameters** because the only operator is passed implicitly, too. You can call these methods either explicitly (as members), or as operators:

```
foo {
    complex c1(2,3), c2(1,1);
    complex c3;

    c3 = c1.operator+(c2);       // a classical method call
    c3 = c1 + c2;                // a real overload
}
```

AleC++ allows overloa of existing operators, but it does not allow for a definition of new operators because that would introduce confusion into the rules concerning association and priority.

4.11. Overload of operator =

Assignment operator (=) can be overloaded as any other operator, but certain restrictions apply. When inheriting (see the section on inheritance), every derived class has to give its own version of the operator, since the definition of the operator is not transferable. Further on, to cover all the applications of this operator you need to define a copy constructor (constructor takes the reference of the source class). These definitions apply to the following cases:

- initialization of objects (e.g. `X x1, x2 = x1;`)
- copying of objects (`x2 = x1;`)

- passing the object as an argument of the function call (transfer by value)
- return of the object using command `return`

4.12. Overload of implicit conversions

You are, by now, familiar with the rules regarding the conversion of operands of different types. The user can add a new conversion to the list of legal conversions, so that the new type (class) functions in the same manner as an inrinsic type. For that, you need appropriate constructors and conversion functions.

```
class X {
        ...
    public:
        X (int);
        operator int (&X);
        X operator+ (X&, X&);
};

X x1(2), x2(3);
int i, *p;

x2 = x1 + 2; // O.K. - 2 is converted into X(2) using constructors
i = x2;      // O.K. - x2 is converted into int using operator int
x2 = x1 + p; // error - types do not agree
```

Appropriate constructors convert common variables into objects of desired type. Conversion of an object into another type requires a conversion function. That function is defined the same way as the operator function, but instead the name of the operator you need to give the name of the target type (it is legal to give a pointer to the type, e.g. `operator int*`). In the example above, `x2` is converted to `int` and the result is assigned to variable `i`. The last line of the example is an error, since the compiler does not know how to add an object of class `X` and the pointer to type `int`.

4.13. Dynamic allocation of memory

Dynamic allocation of memory is known to the users of C. In C++ this feature is raised on the level of a language. Instead of library functions `malloc` and `free`, we have new operators `new` and `delete`. This feature exist in AleC++, too.

4.13.1. Allocation - operator new

Pointers can point to address of a static variable, but can also point to a part of memory allocated under **heap** (free memory whose addresses increase toward to other end of the stack). Allocation can be sufficient for an object of a certain type, as well as for a number of such objects.

```
int *p = new int;               // 4 bytes allocated
int size = 20;
```

```
char *s1 = new char[size+1];   // allocated (size+1)*sizeof(char)
char *s2 = (char *)malloc((size+1) * sizeof(char));   // C style
Point *p1 = new Point (2,3);   // new + constructor call
Point *p2 = new Point;         // default constructor
int *i = new int (5);          // pointer i allocated and set to 5
```

Command `new` is used for memory allocation regardless of type. In contrast to `malloc,` you define a type and not a number of bytes, because the compiler calculates that number automatically. In case you are allocating pointers to classes, you can pass a list of arguments to the constructor, too. You can also set a value pointed by some pointer to a particular value.

Allocation of a vector has a restriction: constructors with arguments are not allowed, only default constructors.

```
Point *vp = new Point [5];
```

Operator `new` can be overloaded to increase the control over allocation. If you do that, you can access the standard (global) operator using access resolution (`::new`).

4.13.2. Deallocation - operator delete

Already allocated pointer to an object needs to be deallocated when leaving the visibility area (in contrast to objects, pointers to objects need to be explicitly allocated using `new`, and deallocated using `delete`), otherwise the memory they point to will not be accessible nor free, which can cause unpleasant consequences.

```
delete i;          // frees the pointer i
delete p1;         // call destructor ~Point, frees p1
delete [5] vp;   // calls destructor for every index of vector vp
                 // and then frees the vector
```

4.14. Inheritance

An analysis of a large number of programs and the experience of programmers revealed that the largest portion of time needed for the writing of a new program is spent on the same routines such as functions for list, stack, tree, graph manipulations. The mechanism of inheritance is a characteristic of object-oriented programming, and it uses these functions as building elements of programs. The essence of inheritance is that “children” inherit “parents” with possible changes. In C++ and AleC++ the children are **derived** and the parents are the **base** classes. There is a mono inheritance (one base class) and multiple inheritance (more base classes). Since derived classes can be base classes for some other classes, the whole system can be very complex as a tree or **hierarchy of inheritance** similar to a family tree.

4.14.1. Inheritance and rules concerning access rights

In a declaration of a derived class, base classes are listed after the derived class name and the colon “`:`”.

```
class A {
        int a1, a2;
```

```
    public:
        int a3;
};

class B {
        int b1;
    public:
        int b2;
};

class C : private A, public B {
        int c1, c2;
    public:
        int c3;
};
```

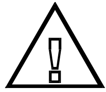
Class `C` is a derived class, which inherits characteristics of classes `A` and `B`. This means that object of class `C` would be 12+8+12=32 bytes. The problem arises with the access rights. The `private` members in the base class are inaccessible in the derived class, and if the class is inherited as private no member (even if declared `public`) is accessible. The situation is somewhat better with the class `B`. Member `b2` is accessible, but `b1` is not. These restrictions can be circumvented f the members of base classes are declared as `protected` instead of `private`. This makes them accessible for methods and friends of the derived class only.

4.14.2. Access to members in the hierarchy bearing the same name

Every class on the inheritance tree has its own area of visibility, which enables the appearance of the members and methods of the same name in many places and their mutual masking. Access to masked parameters is still possible using the operator of resolution:

```
class A {
        int a;
    public:
        void show () ;
};

class B : public A {
        int b;
    public:
        void show ();
};

foo {
        B b;

        b.show();             // calls show of class B
        b.B::show();          // same
        b.A::show();          // show of base class A
}
```

4.14.3. Virtual base classes



Virtual classes are still **not functioning** in the current release of Alecsis.

No class can be repeated in the list of base classes (after the character "`:`"). However, the same class can appear more than once if it was the base class for more than one base class.

```
class A; class B: A {...}; class C: A {...};
class D: B, C {...};
```

Class `A` appears twice in `D` - once in `B` and another time in `C`. If that creates problems, you can declare A as `virtual` class. Regardless of the complexity of the hierarchy, this class can appear only once in the final hierarchy.

4.14.4. Construction and destruction of derived classes

A derived class and its base classes can have constructors. When an object of the derived class type is declared, the compiler calls the constructors of the base classes in the order of inheritance, and then it calls the constructor for the derived class. However if a virtual class exists on the tree, its constructor is called first. The opposite happens with the destructor call - first the derived class, than base, and finally virtual.

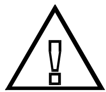
The situation complicates if constructors of base classes require arguments. The arguments are listed in the definition (not declaration) of the derived class constructor:

```
class X {
        int xval;
    public:
        X (int x) { xval = x; }
};

class Y {
        int yval;
    public:
        Y (int y) { yval = y; }
};

class Z : public X, public Y {
        int zval;
    public:
        Z (int x, int y, int z) : X(x), Y(y) { zval = z; }
};
```

Arguments for base constructors are listed after the colon "`:`" and can include all legal expressions. The visibility area after "`:`" includes formal parameters of the derived constructor, source class and global variables. Symbols from those regions can participate in the forming of the argument. If a member of a derived class is also a class, the arguments for the constructor can be passed using the same syntax. Finally, common members of classes can be initialized this way, too. For more details on this subject, consult a manual of C++.

```
class X {
        int xval;
        Y y;            // y has type Y (class)
    public:
        X (int x) : xval(x), y(x+2) {}  // xval is set to x,
                                        // constructor for y
                                        // obtaines argument x+2
};
```

4.15. Virtual functions



Virtual functions are still **not functioning** in the current release of Alecsis.

The mechanism of **late linking** is a fundamental characteristic (if not the requirement) of object-oriented programming. All references to a method and functions are treated as global symbols used by linker in the early phases of the simulation. If the linker is not able to resolve such references the program stops (**early linking**). Contrary to this, **late linking** allows a late decision on the choice of the method, even during the execution of the program (when the method is invoked)

A method needs to be declared as `virtual` in the base class if the mechanism of late linking is to be used. This allows redefinition of that function in a derived class. A redefined function becomes virtual (the word `virtual` is not necessary for it), with the condition that the returned type and the parametric profile is the same as in the base class. **Virtual function cannot be static**, but can be a friend. Global functions cannot be virtual.

Both base and the derived versions have to be defined, or the base function can be declared as purely virtual, but the whole class becomes **abstract**.

```
class Base {
        ...
    public:
        virtual int foo (int);  // base version of foo
};

class A: public Base {
        ...
     public;
        int foo (int);      // redefinition of foo - derived class
};
...

Base b;
A a;
Base *bp;

bp = &a;
bp ->foo(2);                     // calls A::foo
bp = &b;
bp ->foo(2);                     // calls Base::foo
```

Note that the assignment of pointer to a derived class to a pointer of a base class is legal and that the compiler does the implicit conversion. The reverse is not legal without the explicit conversion (**`cast`** operator).

Abstract classes have at **least one purely virtual function**. Those classes serve as the basis for inheriting. It is not legal to define an object, declare a formal parameter, or return the result using return if the abstract class is the type (it is possible with the pointers or references of abstract classes).

```
class object {
        ...
    public:
        virtual void draw() = 0;   // purely virtual function
};
class circle : public object {
        ...
    public:
        void draw() { ... }        // definition circle::draw
};
class rectangle : public object {
        ...
    public:
        void draw() { ... }        // definition rectangle::draw
};

circle c1, c2;          // two objects, class cicrle
rectangle r1, r2;       // two objects, class rectangle
static object *ob[] = { &c1, &r2, &c2, &r1 }; // vector of pointer
        // Assignment of addresses of derived objects is
        // performed using implicit conversion into the base class

ob[0].draw();           // calling circle::draw
ob[1].draw();           // calling rectangle::draw
ob[2].draw();           // calling circle::draw
ob[3].draw();           // calling rectangle::draw
```

If the base class is not abstract, then you need to define both the base and the derived implementation of the virtual function.

5. Basics of simulation in Alecsis

The previous chapters were an overview of concepts already familiar from C/C++, and were meant as an introduction to AleC++. The topic of this and following chapters are constructs of AleC++ not found in C/C++.

Alecsis 2.0 is a hybrid simulator, which means it is capable to simulate both digital and analogue circuits. To state it more generally, Alecsis can simulate both discrete-event and time-continuous systems. This is not a trivial problem, since the techniques of solving these two kinds of systems differ very much. Analogue circuits are simulated using Kirchhoff's laws, thus solving systems of differential equations describing a particular circuit. Digital circuit are simulated using logic states on the inputs, and logic functions determining the output. Where these two types meet A/D or D/A conversion is necessary.

Alecsis is an integrated simulator, which means the algorithms for both kinds of simulation and the conversion process are inseparable. That is, Alecsis is not a case of two simulators "glued" together using special mechanisms of synchronization. AleC++ allows mixing of various analogue and digital elements and constructs, and the user is the only judge of the usefulness of this capability. In this Manual, we will point out the price of using certain constructs in terms of memory, speed, etc.

The simulation engine of Alecsis controls and manages two basic mechanisms, solving of systems of equations represented using sparse matrices, and synchronization of parallel processes and signals in digital components using the *next event* principle and *selective trace* principle. Both mechanisms use the services of the virtual processor, whose role is to execute (interpret) instruction generated by AleC++ compiler and linker. The topology, and characteristics of the circuit is defined by the user regardless of the type of the circuit in question.

5.1. Module concept

When the relative complexity of C++ is considered, the user might get the wrong impression that even the simplest simulation demands hundreds of pages of code,. The example will show what we mean:

```
root module example_1 () {
    resistor  r1;
    capacitor c1;
    vpwl      vin;

    r1  (input, output) 2k;
    c1  (output, 0)     5pf;
    vin (input, 0)      { 0ns, 0v; 1ns, 1v; }

    plot { node input; node output; }
    timing { tstop = 100ns; a_step = 1ns; }
}
```

This is an example of a simple RC circuit with the impulse input of 1V. The impulse is generated using a generator of piecewise-linear voltage `vin`. Besides the topology of the circuit, the `root` module contains the printout format and the data on the duration and the step of the simulation. Such special statements that control the simulation flow can appear in `root module` only. The result is shown in the Figure 5.1:

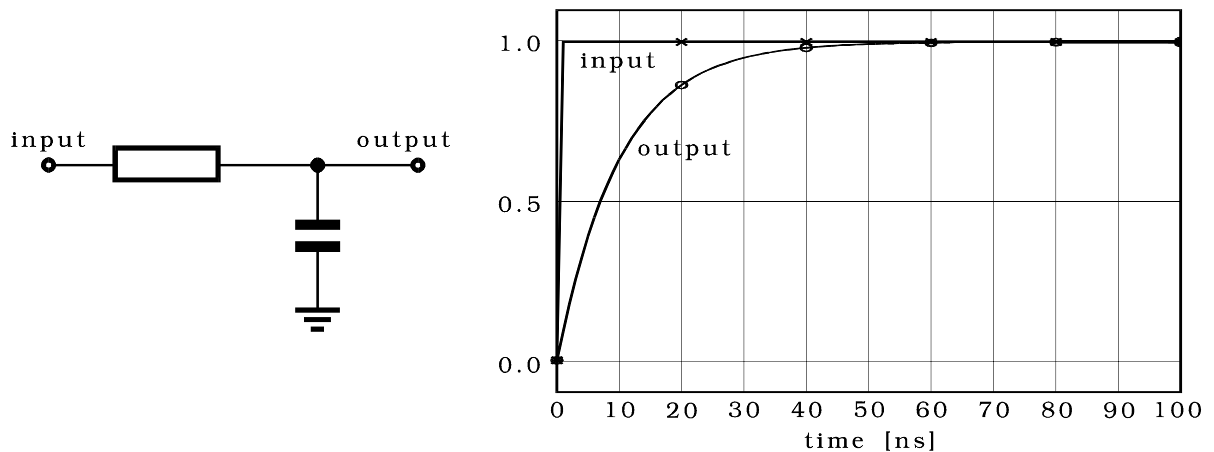


Figure 5.1: The result of the RC circuit simulation

The `module` is the basic element of hierarchy in the hardware description language AleC++. It is used for both discrete-event and time-continuous models, even for A/D (D/A) converters. In C, the execution of a program begins and ends within the `main` function. In Alecsis, the hardware hierarchy begins and ends in the `module` named `root`. Modules have a lot in common with the C-functions, beginning with the syntax, similar interface, to the visibility rules. The similarity is intended since the intention was to keep the spirit of C/C++. On the other hand, modules have something in common with classes as well, since they define new types of components, which can be used evenly with the built-in components.

5.2. Link

Link is the common term in AleC++ defining all types of entities used for connecting components. This term is vague by definition, since it is treated differently for the cases of analogue and digital components. There are five different links: **`node`, `current`**, **`charge`**, **`flow`**, and **`signal`**. These are key words used for link declarations, i.e. they represent link type. Links can be also of user-defined type, i.e. they can be structures, scalars, and vectors, can be initialized, etc. In these characteristics they remind us of variables, and in deed in particular context links can be used in expressions as variables. **However, links are very different from ordinary C/C++ variables, as they are unknowns in the circuit representations, and the simulator engine is used to solve for them.**

The first four mentioned link types (`node`, `current`, `charge`, and `flow`) are analogue. They are used for description of analogue components, and appear as unknowns in the system of ordinary differential equations. The value of the link is evaluated by solving the system of equations. These four link types have the same implementation, but differ in their physical meaning. `Node` is the basis for the analogue part of the simulator, since it represents the physical link - conductor, and because the system matrix is formed using nodal analysis method. The variables of link type `node` are actually representing the node voltage. This method is expanded in Alecsis (as well as in all well-known simulators, including SPICE) to include the elements of zero resistance (voltage sources, inductors, etc.) Such an element generates new (branch) equation, and the branch `current` appears as the variable. `Charge` is used for nonlinear capacitance modelling, to allow separate discretization and linearization of the model. `Flow` is a keyword for a general analogue link, whose physical meaning is not given in advance. This separations of analogue circuit unknowns is due to physical differences, in order to control the convergence tolerances separately. The absolute tolerance for flows is not given in advance, but can be defined by the user. When used in expressions, referencing the name of the analogue link relates to its value obtained in the last solution of the system of equations.

Signals differ significantly from other link types. Signals are links that carry some logic value. Digital systems in Alecsis are modelled as parallel processes. Signals are used for synchronization, too. Signals can be read-only, write-only, or both. Every signal has associated memory according to link data type, which contains its current value. That value is obtained if the signal name is used in an expression.

If the link type is `signal`, the link by default has **digital aspect** if not explicitly defined with **analogue aspect**. If a link is connected to a component whose fundamental type (A or D) does not agree with the link aspect, the component has a dual, or hybrid aspect. Links wit hybrid aspect make the simulator generate A/D (D/A) converters.

Note: The link declaration has the link type and the link data type. The link type can be `node`, `current`, `charge`, `flow`, `signal`, or some composite type made of these basic types. The link data type refers to type of its value (i.e. variable type - `double`, `int`, enumeration type, ..). The following declarations are legal:

```
node double X; signal three_t Y;
```

There are default link data types. It is clear that the `node` or `flow` would have value of type `double`, so there is no need to mention that explicitly. However, for signals it is usual to state the type explicitly, as it is normally some enumeration type.

5.3. Module declaration

Modules have prototypes in much the same way as functions do. Module has architecture, and interface for communication with the surroundings. The prototype declaration is related to its interface. The declaration allows the compiler to check the agreement of actual with formal links on when the component of the module type is referenced.

prototype_module:

module *<flow_sc> <library_name* **.**>*module_name (<list of formal links>)*

flow_sc:
current
charge
flow

list_formal_links:
formal_group
list_formal_links **;** *formal_group*

formal_group:
< link_type > <type> <direction> list_links

list_links:
link
list_links **,** *link*

link:
link_declarator <= constant_expression>

link_declarator
identifier
link_declarator [<vector_dimensions>]

link_type:
signal
node
flow_sc

direction:
in
out
inout

Module has its name, which can be used evenly with the names of built-in types of components, such as `resistor`, `capacitor`, etc. Modules can return the link. If the link type is missing, type `node` is inserted by default, except in the case of the enumeration link, where `signal` is default. If the link data type is missing, the default for analogue links is `double`, and for digital `int`. Direction refers exclusively to signals, and is `in` by default. Initial values can be defined for signals as an option.

```
typedef enum { 'x', '0', '1' } digital3;
struct Net { digital3 send, recv, ack; };

module X (i, j);                      // two node links - i and j
module Y (digital3 in a, b; digital3 out y); // signals
module Z (signal digital3 in a[]="001"; signal Net out network);
module current misc.M (node i, j);    // library "misc"
```

These examples of module declaration encompass all of the rules. Module `Z` contains the declaration of the first formal link `a` as a digital vector, whose dimensions are not defined. As with functions, the dimensions of the

vector will be defined when the actual vector comes to that position due to the module connecting. By default the dimensions of the vector is 3, based on the initialization expression - enumeration string `"001"`. Module `M` expects two nodes `i` and `j`, and returns the current under its name. Explicitly defined library (`misc`, and separator `.`) instructs the linker to look for the body of the module in the mentioned library (without searching all of the available libraries). If more libraries are visible, all containing a module with the same name the concept of the explicitly defined library will solve the dilemma.

Formal links do not represent unique entities, just references to real links on the module interface. These declarations are usually found in header files included by using command `include`. More than one declaration for the same module can be found in the same file with the condition that all those declarations overlap, or append each other.

Note: Module definition is valid also as a declaration for the following code, which is why carefully placed definitions can render declarations unnecessary. However, if modules are compiled, and stored in libraries, prototypes (declarations) of modules are necessary and have to be added using statement `include`, if they are gathered in one header file.

5.4. Module definition

Declaration of a module is helpful to the compiler during numerous checks. The definition of the module gives its actual content. The definition of a module can be compiled and stored in some library, since modules are external units, that is they are subject to linking operations.

Modules can be defined on the global level only, and cannot be nested. Note that more than one definition of a same module in same file is an error, which was not the case with declarations. Definitions repeat the declarative part, but also contain the body of the module. The body of a module is bounded by characters '`{`' and '`}`'.

The module architecture depends upon the modelling approach of a particular electronic component. The component can be digital, analogue, or hybrid. It can be represented structurally, as a collection of connected components, functionally, using constructs based on commands, or using a combination of both approaches. For each approach, a region exists in the body of the module where those constructs are allowed.

module_definition:
module_declaration { module_body }
module_body:
<declarative_part > < topological_part > < functional_part >

The body of a module can have all three, or none of the mentioned regions, but the former case has no practical implications.

5.4.1. Declarative part

The body of a module may contain some entities (links and components) that are local, i.e. used only inside that module. AleC++ needs them declared before they are used, which is a rule from the languages making its basis. Visibility of local links is the same as with local variables, being the body of the module, with the notion that local links duration is the whole the simulation (like static variables).

The syntax of the declaration is similar to the one for the interface, just the direction is not stated. Direction is by default `inout` for signals, and does not make sense for other links.

```
signal digital3 matrix[][4] = { "0011", "011x", "1101", "10xx" };
node vdd, vss;
digital3 clock = '0', mask[3:0] = "0010";
```

As with the variables, signals-vectors can be initialized and can have inverse dimensionality. If link data type is enumeration type, the link type can be omitted, as it must be signal. If the link data type is left out, defaults are the same like for formal links: `int` for signals, and `double` for others.

There are cases when local links need not be declared before they are used. If an undeclared name appears during the component connecting, a (scalar) `node` of the same name is implicitly defined, with the same characteristics. This rule makes it easier to use Alecsis for the users of SPICE, which does not demand nodal declarations.

Component declaration introduces names, and links them to the component type. The declaration consists of the type of the component, and the list of elements which can be used for modelling.

```
resistor r1, r2;
capacitor cload, cs;
module rsff ff1, ff2;
rsff ff3;
module ttllibrary.ff3;
module frsff (digital3 in reset, set; digital3 out q, qbar) ff4, ff5;
```

If the type of the component is module (i.e. not a built-in Alecsis component) it can be cited with or without the key word `module` (as with structures/classes, the key word needs to be used if the name `rsff` is masked). It is legal to give the name of the library (the row before the last), and even the complete declaration (the last row). If the complete declaration appeared somewhere in the previous text, it is sufficient to give the name of the module for the name of the component.

There is a possibility to skip the component declaration, again to make Alecsis comfortable to SPICE users. The compiler can, based on the name of the component, determine the type (see the paragraph on `implicit` constructs).

5.4.2. Structural part

We will focus on the component connecting and the definition of their parameters in this section. To connect components, we list their names and actual links, and perhaps the values of the parameters:

```
r1 (n1, n2) value = 2k;
r2 (n2, 0) 3.3k;
cload (load, 0) 15pF;
```

The syntax of connecting is:

component:
    name (<list_actual_links>) parameters

list_actual_links:
    static_link
    list_actual_links , static_link

static_link:
    name_of_link

static_link [constant_expression]
static_link [constant_expression **:** *<constant_expression>]*
static_link **.** *structure_member*

parameters:
;
paremeter_value ;
{ list_parameters }

list_parameters:
parameter_in_list
list_parameters_of_ parmeter_in_list

parameter_in_list:
parameter_name= parameter_value ;

parameter_value:
constant_expression
parameter_name = parameter_value

Actual link can be a scalar, but also a composite link type (vector, structure ...). The link of composite type can be for used for connecting as a whole, by citing its name, or one can use only some of its members. Indexing can be used if the link is an array (vector, matrix, etc). The notion of a **static link** is applied either to the link, or to any of its parts, which can be fully determined during the compiling. For that reason, it is necessary that the index is a constant expression. We will return to static links when further describing syntax of AleC++. If the link which is an array is indexed using two expressions and character ':' between them, its dimensionally is not changed, but its boundaries are changed (reduced). The result of this operation is called **slice**.

```
signal digital3 s, v[10], m[3][4], v2[15:0];
signal Net data;
...
c (s, v[1], v, m, m[2], m[2][1], v2[9:7], data, data.ack);
```

In this example, component `c` is connected using 9 links: scalar `s`; scalar from the position `1` of vector `v`; vector `v` of length 10; matrix `m` of length 3x4; vector of length `4` from the position `2` of matrix `m`; scalar from position `2,1` of matrix `m`; slice of vector `v2` from position `9` to `7` (still a vector); structure `data`; and a scalar `ack`, member of the structure `data`. The compiler is responsible to determine if this list of arguments agrees with the prototype of module of component `c`. From the syntax standpoint, this is a legal list of actual links.

Alecsis is equipped with a few, but carefully chosen set of predefined (built-in) components. These are fundamental components, which are often used in electronic circuit. They can be also used as the basis for modelling of other components (you can find this explained in detail in the chapter on analogue simulation). The number of component parameters varies from one component to another. Simpler ones take only a single parameter of `double` type under the name `value` (resistor resistance, coil inductance, etc). In this case only a numerical value can be given. If there are more parameters, their list needs to be bounded by parentheses.

```
cload (load, 0) 15pF;
cs (n2, 0) value = 10pF;
mos1 (n1, n2, 0, 0) { model = nmos1; l=2u; w = 3u; }
```

Components of type `module` can have parameters, too. Their syntax does not differ from the one for built-in components, however the declaration needs to be expanded to include the names, and types of legal parameters. We did not talk about this up to this point, since this goes into functional simulation.

5.4.3. Functional part -- action block

Electronic component (subcircuit) can be defined functionally in several ways:

- By defining a logic function of the component . This is purely functional approach for the digital aspect.
- By stating the model equations. They contribute directly to the system of equations for the whole circuit (system). This is purely functional approach for the analogue aspect.
- By defining and equivalent circuit topology, and calculating the parameters of the components in that equivalent circuit in AleC++ code. For instance, nonlinear model can be represented as equivalent linear circuit whose parameters are changed in every iteration. This is a combined approach for the analogue aspect.
- By combining all approaches given above - combined approach for hybrid aspect.

To realize this functional description, we need a part of the module body where C/C++ -like code can appear. A region of the module body beginning with the keyword **`action`** is used for that. This functional region is bounded using parentheses, creating a narrower visibility area, which allows masking of elements, local and formal links, etc. If our module accept parameters, they are actually accepted by the `action` block. For that, in module definition, you need to give a list of `action` parameters as if a prototype of a function is created:

```
module X () {
    ...
    action (int n, double p, char *name="initial value")
    {   /* the code describing fucntional description */   }
}
```

As was the case with function parameters, action parameters can have default values. We recommend this approach, since it allows the correct work of the component even if the parameters are not set during connecting.

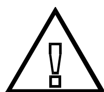
Absence of `action` parameters can be signalled by type `void`, parentheses without parameters '`()`', or by leaving out parentheses after the `action` keyword. Action can be defined with a variable number of parameters using symbol '`...`' as it is defined in the chapter on functions.

Action parameters are added to module declaration:

```
module and2 (digital3 in a, b; digital3 out y)
                       action (double tplh=10ns, double tphl=10ns);


module X () {
    digital3 s1, s2, s3;
    and2 a1, a2;

    a1 (s1, s2, s3) { tplh = 11ns; tphl = 13ns; }
    a2 (s3, s1, s2) action (11ns, 12ns);     // alternative method
}
```

As you can see in the example above, components that have `module` type (not built-in) can utilize another method of parameter setting, too. It resembles function calls, because a list of arguments follows the word `action`. Linking of parameters and arguments is done **by position**, which is different from the commonly used **associative method.** As was the case with the functions, parameters with initial values can be left out.

Note: **Action parameters** can be used bidirectionally, i.e. they can return the value. They **are passed *by reference***, unlike parameters of C function, which are passed *by value*. For one application of that feature see explanation of `plot` command in this Chapter.



If, for instance, `action` block in some module expects parameter of type `double`, and you connect that `module` in the parent `module` with the appropriate `action` parameter as integer constant, the simulator will not issue any warning, but will not give you expected results. The reason is that *an implicit conversion is performed*, since the parameter is passed by reference. This is, however, usually not what you wanted. **Be careful with `action` parameters -- always pass the constant or variable parameter of correct type.**
In the following versions of Alecsis, warning will be issued for such cases.

Note: Action parameters of one component can be accessed from other components using indirection operator `->`. For instance, if `module X` in the example above has its own `action` block, inside that `action` block you can access action parameters of component `a1` as `a1->tplh`, `a1->tph`. (component name behaves as the pointer to the structure comprising of its `action` parameters). For another application of that feature, see section on `plot` command in this Chapter.

All variables local to the action area last throughout the simulation, and are therefore static. This is important, since `action` block can be executed many times during the simulation run, and the variables must not be reset. However, each separate component has a separate memory, different from the memory for other components of the same type. If needed, some variables can be defined `static`. These variables are similar to static members of classes in that memory reserved for them is common **for all components** of the same type in the circuit. This means change to the value of that variable in one component will ripple to all other components of the same type. Since this allows communication using the "back door", circumventing action parameters and links, great care needs to be exercised in order to avoid unwanted effects.

`Action` block is where functional behaviour of the `module` is defined. If the `action` block is left out, `module` is a set of components connected in a desired way. In that case, `module` represents only a subcircuit, and is used to describe the whole circuit hierarchically.

5.4.4. Modelling of parallel processes

It should be noted that all components in a circuit (system) are active simultaneously, or, in programmers' terminology, in parallel. To enable Alecsis users to describe that parallelism, and to control execution of parallel blocks, we have introduced `process` into the functional description. Processes are described inside the `action` region.

The `action` region is divided into the declarative part and the command part.

action_region:

action *<trigger_sc> <(<parameter_declaration>)> <action_body }*

action_body:

<declarations> <commands>

commands:
command_list
process _list

process_list:
process_command
process_list process_command

process_command:
<process_name :> **process** *<sinhro> { <commands> }*

sinhro:
trigger_sc
(sensitivity_list)

trigger_sc:
structural
post_structural
initial
per_moment
post_moment
per_iteration
final

sensitivity_list:
static_link
sensitivity_list , static_link

The `process` is the backbone of functional modelling in AleC++. The processes are given in the `action` block, and they consist of commands. Those commands are executed during the execution of the simulation. The processes ought to be synchronized.

In discrete-event simulation, signals synchronize processes. The **event** has happened when a state (value) of a signal is changed. All processes sensitive to that particular signal become active when such event happens (commands of the `process` are executed). After that, processes sensitive to that signal are in inactive state, or **latency,** until new event happens. This agrees with the concept of logic states, events, and transfer of signals throughout the circuit in digital (discrete-event) simulation. A `process` can be made sensitive using the sensitivity list, which lists all signals whose change can activate the `process`. As was the case with actual links, we can list signals, signals with indices, or, if the signal is a structure, a member of the structure. Process is activated if a change occurs **on any** of the listed signals. In the case of a composite signal, an event has happened if an event has happened on at least one of its scalar elements (positions of a vectors, or members of a structures).

Sensitivity list must be avoided if the process has **`wait`** command (see the section on the `wait` command).

If the `process` does not have a sensitivity list, `wait` command, and is not synchronized in any other way, it becomes a world on its own, and is useless as far as simulation is concerned.

More than one `process` can be sensitive to a particular signal. When an event happens on the signal, these processes are executed by the simulator one by one, but they appear as parallel from the point of view of circuit functionality.

In analogue circuit simulation, different synchronization mechanism needs to be implemented. Analogue components are active all the time, since they contribute to the system of equations that is repeatedly formed and solved during the simulation. As Alecsis is used for time-domain simulation of nonlinear circuits, there are two loops: the time-loop (outer), where simulation is executed in many discrete time instants; and iterative loop (inner), where, in every time instant, nonlienar circuit is solved iteratively. From that point of view, no synchronization is necessary. However, synchronization of processes is very useful in analogue simulation, too. If a component is linear and time-independent, as resistor, there is no need that it is executed in every time-instant, and in every iteration. The appropriate `process` can be executed once, before the actual simulation starts, which saves CPU time. If a model is linear, but time-dependent, the `process` can be synchronized to be executed in every time-instant, but out of the iterative loop.

For synchronization of analogue processes, some internal synchronization signals are generated and controlled. The value of these special signals cannot be accessed in expressions, it can be used for `process` synchronization only. These signals are:

- **`structural`** - activated before the simulation, during the creation of a hierarchical tree, that represent the circuit hierarchy in simulator memory. It is intended to be used for processes that contain command **`clone`**, that creates array of components (or subcircuits). As the command `clone` results in a change of the circuit structure, it has to be executed before the simulation starts, and before the circuit hierarchy is formed in the simulator memory.
- **`post_structural`** - Activates before the simulation, after the current hierarchical level is complete. It is used for modification of signal attributes (see section on user-defined attributes).
- **`initial`** - Activates only once, at the beginning of the simulation, when time $t=0$. Usually used for some intializations, but also to calculate contributions to the system matrix that are not changed during the simulation, i.e. that need not to be calculated inside the time loop and the iterative loop.
- **`per_moment`** - Activates in every new time instant of the simulation $t=t_{n+1}$ before solving the system of equations. It is intended to be used for modelling of linear but time-dependent contributions to the system matrix (e.g. linear capacitors, linear time-dependent voltage or current sources, etc.). Usage of link name in expressions in such `process` returns its value from the previous (last solved) time instant ($t=t_n$)
- **`post_moment`** - Activates in every time instant after reaching the solution of the system of equations. Processes, which need solution from the moment $t=t_{n+1}$, use this synchronization. If a circuit consists of digital elements, and links, solving the system is unnecessary, so this and the previous synchronization signal are activated simultaneously.
- **`per_iteration`** - If the circuit has analogue elements, this synchronization signal activates in every new iteration before solving system of equations. If there are n time instants, and m iterations in every time moment, processes sensitive to `per_iteration` signal are active m x n times. Used for nonlinear analogue elements, where contributions of the linearized model to the system of equations are calculated in every iteration, until convergence occurs. If the circuit does not have analogue elements, these processes will not activate.
- **`final`** - Pair with the **`initial`** signal - activates only once, at the end of the simulation. If a file is opened or memory is allocated in the `process initial`, this is the place where everything needs to be closed and ended.

Some of the above mentioned signals can be very useful in digital applications, too, especially at the beginning or an end of the simulation.

All legal AleC++ commands except **return** can appear in a process, since the process cannot terminate (the whole simulation can be terminated using exit, but this a very rough solution). Command **continue** needs a small modification of standard rules; considering the cyclic nature of a process: usage of this command outside a loop means a jump to the first command and restart of the process. If the process is sensitive to signals this causes halt of the process until an event occurs on signals. You cannot jump from a process to a process using command **goto**.

Every process creates a separate visibility area where local objects can be declared. All initializations of such object have to be static, that is have to contain constant expressions. Process objects, explicitly declared as static using key word static, are common for all copies of the process that are created by declaring more components of the given type. The rest of the objects, not explicitly declared as static, are unique for every copy of the process.

There can be more than one process in an action block. If there is only one process in an action block, the keyword process can be omitted. In that case, synchronization signal is given after the keyword action. If no synchronization is defined, per_iteration is used by default.

```
module X (digital3 in x, y; digital out z) {
    action (double delay) {
        int shared;       // common variable for all processes in
                          // this action block
        static nmodules;  // common for all components of type X
        p1: process initial {
             // initial activities
             ...
        }
        p2: process (x,y) {
             // activation on event on signals x or y
             ...
        }
        p3 : process final {    // names p1, p2, p3 can be omitted
             // final activities
             ...
        }
    }
}

module Y (i, j) {
    resistor r;
    capacitor c;
    action per_iteration {
        // ... an analogue process
    }
}
```

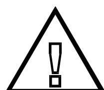
5.4.5. Variable number of action parameters

The number of parameters in the header of the action block of the module can be variable. As action header resembles the header of the function, the technique for variable number of parameters is the same as for the C/C++ -like function, described in Chapter 4.

```
#include <varargs.h>

module ResistorWithTemperatureCoefficients (node i,j) {
   resistor res;
```

```
    res (i,j);

    action post_structural (double resistance, ...)  {
       char *args;
       double tnom = 300., tc1,tc2,tce;
       tc1=tc2=tce=0.;

       va_start(args, resistance);   // setting pointer args

       if (tc1 = varargs(args, double))  // read other parameters
          if( tc = varargs(args, double))
             tce=varargs(args, double);

       if (tce)
          res->value = resistance*pow(1.01,tce*(temp-tnom));
       else if (tc1)
          res->value = resistance*(1+tc1*(temp-tnom)+
                                 tc2*(temp-tnom)*(temp-tnom));
       else
          res->value = value;
    }
 }
```

Parameter `temp` in the example above is the user-defined temperature (see description of simulation `options` in this Chapter).



There are some differences in storing function formal parameters and `action` parameters. For that reason, alignment of parameters of type `double` using flag `DWORD_ALIGNMENT` is not necessary for `action` parameters, even for the computers that need that for C/C++-like functions. Moreover, you should be sure that this flag is not defined when file `varargs.h` is included

As we have pointed out for functions, functionality of macros defined in Alecsis `varargs.h` file depend on processor. Therefore, if you install Alecsis on computer that was not predefined in `Makefile`, some adaptations on varargs.h might be necessary.

5.5. Implicit declaration of components

SPICE deciphers the component type from the first few characters in the name of the component. All components in Alecsis need to be declared before connected. For some types of common components this can become rather tedious, and is not convenient for SPICE users. For that reason, we have enabled such implicit declarations in AleC++, but only as an option. Construct `implicit` is used to fulfil that.

implicit_command:
    implicit *{ association_list }* <;>

association_list:
    implicit_association
    association_list implicit_association

implicit_association:
component_type list_names ;

component_type:
built-in_type
module_declaration

This `implicit` declaration defines one ore few characters that represent beginning of the component name. These characters are normally association of a component type. After such a declaration, all the components, whose names begin with these characters, have a declared type, and do not have to be explicitly declared. The name of the component has to be at least one character longer than the implicit symbol. In case of a conflict (two, or more suitable symbols) the longer is chosen. This command can be used many times in the text, but only on the global level. The care needs to be exercised not to make unwanted redefinitions.

```
implicit {
    resistor r, R;
    capacitor c, C;
    mosfet m, M;
    bjt q, Q;
    module rsff ff;
    rsff rsf;
}

module X (i, j, k, l) {
  r1 (i, 0) 2k;         // O.K. - resistor
  R2 (j, 0) 4k;         // also
  m1 (i, j, 0, 0) ...; // MOSFET
  ff (i, j, k, k);    // error - the name is the same as the symbol
  ff1 (i, k, k, k);     // O.K. - flip-flop
  rsf1 (i, j, j, l); /* resistor or rsff? - rsff, since it
                        is longer! */
}
```

5.6. The root module

Every hierarchical tree has a root. In AleC++, description of a hierarchy of an electronic circuit begins from the module named **`root`**. In this case, keyword `module` is not necessary. Everything said about modules is true for the `root module`, with certain modifications and additions. The `root module` must not have formal signals and/or `action` parameters, since due to its position on the top of the tree it cannot receive any. In the same fashion, it cannot return under its name any link. Finally, the `root module` has three additional constructs defining the conditions of the simulation and printing its results:

- **`plot`** - for printing out the results;
- **`timing`** - for timing control;
- **`options`** - for defining simulation conditions (e.g. tolerances).

Note: Commands `plot`, `timing` and `options` are placed between structural and functional part of the `root module`. In this space, these three commands can be given in any order.

Note: In the `root module`, functional part can be omitted, but the structural part must be described. Command `timing` cannot be omitted. Command `plot` can be omitted (but it does not make to much sense to simulate without printing out results). Command `options` can be omitted.

5.6.1. Print control -- command plot

Alecsis does not have waveform display capabilities that can be used to view the results of simulation. For that, separate program Agnu is used. Alecsis creates an output file, with the results of the simulation in numerical form. There is not much sense in saving states of all digital signals and values of all analogue variables in the circuit, because that would make the output file too big to handle. Only signals and variables specified by the user are saved. For that, the command **`plot`** is used. Alecsis creates the output file during the simulation. If the simulation is terminated for any reason before the final time-instant is reached, all time-instants solved until that moment are saved. The output file carries the same name as the input file, with the extension **`.ar`** (Alecsis results). (Input file has extension `.ac`.)

printing:
  plot *{ content_of_printing } <;>*

content_of_printing:
  element_of_printing
  content_of_printing element_of_printing

element_of_printing:
  caption *constant_string ;*
  link_type <type> <direction> link_list;
  sweep *<type> link;*

link_list:
  link
  link_list , link

link:
  static_link < (element) >
  absolute_path /static_link <element>
  identifier body_function

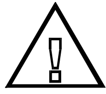
absolute_path:
  element_name
  absolute_path / element_name

Note: Keyword `plot` can be replaced with `out`, for compatibility with earlier versions of Alecis.

The links are specified with their link types. The link type comes from the set of legal types (`node`, `signal`, etc.). The link data type (`double`, `int`, etc. or some composite type) is not necessary for link local to the `root module`, but if you want the value of link situated somewhere else on the hierarchical tree, which is a vector, structure, or a digital signal, the link data type is necessary.

For link that is not local to the `root module`, the absolute path is given as the part of its name. The path is composed similarly to the path in UNIX operating system. The absolute path is the path from the `root module` to the internal link via names of the components making the path (i.e link `X` in submodule `Y` is given as

`Y/X`). The link itself can be an identifier, identifier with index (for arrays), or reference to a member of structure, by rules of static links. If the name is a composite link, the printout will consist of all its elements.



As said above, link data type has to be given for vectors, structures or signals that are not declared in the `root module`. If vector of nodes `U` of length `N` is declared in submodule `Y`, command:

```
plot { node Y/U; }
```

results in printing only the first vector position `U[0]`, instead of the whole composite link, i.e. all `N` positions. The compiler does not see the declaration of vector `U`, which is lost after parsing the `module Y`. For that reason, `U` is treated as the scalar node, not as the vector. The problem can be solved if a vector type is defined on the global level, before the `root module`:

```
typedef double tmpvec[N];
```

and printing is performed using:

```
plot { node tmpvec Y/U; }
```

In this way, link data type is given explicitly, and the compiler knows that `U` is a vector.

The title to be passed to the program for graphical presentation can be controlled using keyword **`caption`**. If `caption` is omitted, the name of the `root module` is used as the title.

The variable on the x-axis is by default the time in seconds. However, any circuit variable can be set on the x-axis using keyword **`sweep`**.

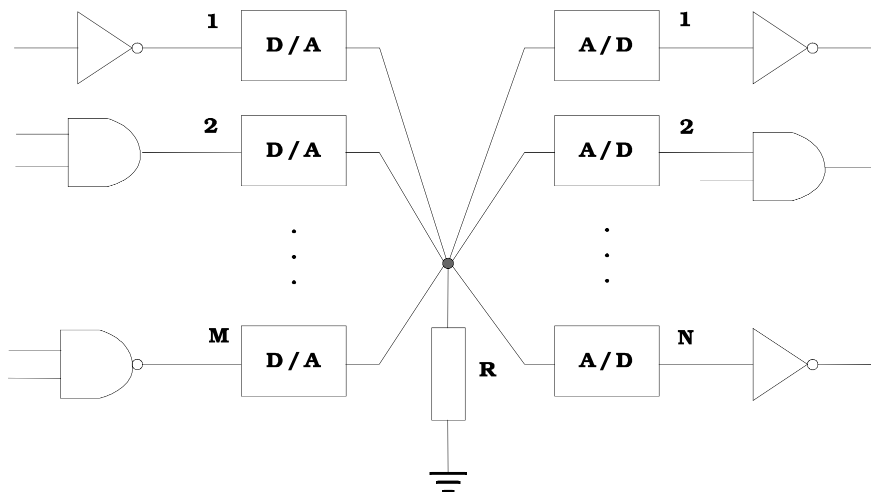


Figure 5.2: Problem of link with the hybrid aspect is solved by inserting converters

A link can have hybrid aspect if it is connected both to analogue and digital components. The case depicted in Figure 5.2 is a general one, since link is connected to analogue components (no matter how many of them) to *M* outputs of logic gates, and to *N* inputs of logic gates. In this case, Alecsis automatically generates *M* D/A converters and *N* A/D converters, as shown in Fig. 5.2. That means, converters are generated for every digital circuit, which is connected to an analogue link via input or output.

Such a link has an analogue aspect (unique analogue value), as *N* digital aspects, viewed from *N* A/D converters, and *M* digital aspects, or viewed from *M* D/A converters. We can print out all the aspects of such link. Analogue aspect is obtained by specifying the analogue link type (node, flow, etc). By specifying the direction indicator as in, and the link type signal, we get *N* solutions from the inputs of A/D converters, that is if the indicator is out - *M* solutions from the inputs of D/A converters.

```
struct S { three_t send, recv; };
root module test () {
   vgen vg;
   signal three_t v1, v2[4], v3[3:0]="0010";
   node n1, n2, n3, n4;
   module X x;
   signal S s1, s2;
   ...
   plot {
     caption "results of simulation of root module test";
     node n1, n2; //drawn in the same group - common scaling
     node n3; node n4; // drawn separately
     signal v1, v2[2], v3; //signals are always drawn separately
     current vg;          // current trough the voltage source vg
     signal three_t x/y/z/data;
     signal three_t in v1;   // all values           - A/D
     signal three_t out v1(e1), v2;  // all drivers  - D/A
     signal S s1, s2.send;
   }
}
```

Nodes n1 and n2 will be shown in the same group, which means they use the same scaling for y-axis. Nodes n3 and n4 are stated separately, and will be shown on separated waveforms. Signals v1, v2, v3 are also given separately, one beneath the other, which is always the case with digital signals. The whole of the signal v3 will be shown (from v[0] to v[3]). Since vg is a voltage source, compiler has generated the current of the same name flowing through the source (as in SPICE), which can be accessed using the keyword current. The next line defines the printing of the signal named data, which is reached using the listed path, as it is not on the root hierarchy level. In this case compiler accept the given link data type of the signal data, as the information about the real type is lost after parsing module z, where this signal is local (when preparing the data for simulation, simulator checks whether the signal data really exists). Supposing that some analogue components are connected to v1 and v2, the following line enables us to get all results of A/D conversion for v1. After that the results of D/A conversion for v2 are demanded, as well as the results of D/A conversion for v1, but only for the component e1. The last line defines the printing of all members of signal-structure s1, and member send of signal-structure s2.

In some cases, we do not need a value of the link as the simulation results, but the result of some computation with that values (e.g. difference of two node voltages, their ratio, etc.). In this case, instead of the link name, the body of some function that performs the computing is given, and the result is returned using command return.

```
plot {
       node double power { return (n1-n2)*vg; };
     }
```

In this way, the result named power, calculated as the difference of node voltages n1 and n2 multiplied by the current vg, is given in the output file. One of the two type declarations - of the link type (node) and the data type (double) - can be omitted, but not both of them. The declared data type must agree with returned type.

Note: It is more consistent to use only data type declaration (double), as power is a new variable, not a new link.

In the above example, computation is simple, but the user can implement more complex function bodies. The example can be rewritten as:

```
plot {
       node double power { double node1, node2, node_diff;
                           node1 = n1;  node2 = n2;
                           node_diff = node1-node2;
                           return node_diff*vg; };
     }
```

The problem arises with the links that are not on the root hierarchy level, i.e. that are not local to the `root module`. The symbol '`/`' used to define the path through the hierarchy is used for division in expressions, and would be understood as such in function body. Therefore, computation can be performed only with values of links local to the `root module`. That means, all links to be used in computation are to be declared in the `root module`, and than passed to submodules when they are invoked. The another way around is to have that links declared only in submodules, but to return their current value using `action` parameter. As already said in section on `action` block, **parameters of the `action` block can be used bidirectionally, i.e. they are passed by reference, unlike parameters of C functions;** and they can be accessed using indirection operator, i.e. **name of the component behaves as a pointer to the structure that comprise its `action` parameters.**

```
module Y (...) {
   node n1;
   action (double p1=0;) {
   ...
   process per_moment {
       p1=n1;
   }
}

root module X {
   Y y;
   y(...);
   ...
   plot { double n1_value { return y->p1; }
}
```

5.6.2. Timing control

timing:

timing *{ time_control }* *<;>*

time_control:

setting
time_control setting

setting:

parameter = timing_rhs ;

timing_rhs:

constant_expression
parameter = timing_rhs

This construct controls time parameters of a simulation. Simulator recognizes the following parameters.

Table 5.1. Parameters for simulation time control.

Name	Default value	Meaning
tstop	/	duration of the simulation (in seconds)
a_step	/	starting time step of analogue simulation (in seconds)
a_stepmin	a_step/100	minimal allowed time step of analogue simulation (sec)
a_stepmax	min(a_step*100, tstop/100)	maximal allowed time step of analogue simulation (in seconds)
tprint	0	printing step (in seconds)

For example:

```
timing {
    tstop=10ms;
    a_step=0.1ns;
    a_stepmax=0.1ms;
}
```

Of all these option only tstop applies to digital simulation. Digital simulator advances using the next-event technique. It performs the simulation only when an event happens The result of that simulation are new events, scheduled to happen in some future time. After that, the simulation time advances to the time of the next scheduled event. For that reason, time step does not exist for digital simulation - it jumps from one event to another. The results are printed for every event in output file, so the tprint parameter does not have effect, too. Digital simulator performs the simulation for all events scheduled before tstop. If there is no event scheduled exactly at time *t*=tstop, the simulator repeats the printout of the last state for the time *t*=tstop in order to complete the waveforms for the graphical presentation.

For analogue simulation, parameter a_step is obligatory, too. This value is used just to begin the simulation, since simulator alters the time step during the simulation, according to the dynamics in the circuit. Time step is chosen to have the maximal allowed value (to save CPU time), so that the accuracy of time-domain simulation is within limits determined by the tolerances abs_LTE and rel_LTE (see the following section).

The time step is kept in limits (a_stepmin, a_stepmax). Parameter a_stepmax is used to avoid to big time steps that makes waveforms, although accurate, to look discontinuous. The default value is hundred times bigger than the given a_step, but a_stepmax cannot be bigger than t_step/100. Parameter a_stepmin enables us to avoid too small time steps. Time step appears as denominator in reactive component models, and to small value can create numerical problems. Besides, if the time step is too small, the simulation can last very long, and the reason might be unimportant - for instance, rapid change of voltage on some parasitic capacitances. For that reason, it is useful to limit the smallest value of time step. However, if the simulator reaches value a_stepmin, the simulation error is not within given tolerances. The simulator issues warning message suggesting decreasing value of a_stepmin or increasing tolerances for numerical integration abs_LTE and rel_LTE (or, in case the circuit contains ideal switches, tollerances SC_vtol, SL_itol, or SDDT_tol).

Value of parameter `t_print` gives the minimal time difference of results printed in output file. It is very useful to limit the number of printed points on waveforms, since to big number of points results in very large output files, and often does not contribute to the readability of results. Default value is 0, when all computed time points are printed out.

5.6.3. Simulation options

Options control only the analogue aspect of the simulation. Here, many parameters that control the simulation run can be set. Clearly, this command is optional, as all these parameters have default values. We can divide this set of parameters in four groups:

- control of numerical integration;
- control of iterative process;
- control of sparse matrix solving;
- control of component models;

5.6.3.1. Control of simulation time (numerical integration)

Table 5.2. Control of numerical integration.

Name	Default value	Meaning
method	Gear2 (2)	The method of numerical integration: can be None (0), EulerBackward (1), or Gear2 (2).
abs_LTE	1.0e-12	Absolute tolerance of local truncation error (LTE)
rel_LTE	0.001	Relative tolerance of local truncation error.
SC_vtol	1mV	Accuracy of voltages of switched capacitors.
SL_itol	1uA	Accuracy of currents of switched inductors.
SDDT_tol	1000	Accuracy of quantities that are numerically integrated using eqn command if switches exist in circuit.

An example of statement `options` is:

```
options {
        method = EulerBackward;
        abs_LTE = 1.0e-11;
        SDDT_tol = SC_vtol = 0.01;
}
```

Note: Values of parameter `method` - `None`, `EulerBackward` and `Gear2` are actually integer values, defined in standard Alecsis header file `alec.h`. In that file, it is defined:

```
#define    None            0
#define    EulerBackward   1
#define    Gear2           2
```

Therefore, to use textual values of parameter `method`, you should have file `alec.h` file included before your `root module` definition, using command:

```
# include <alec.h>.
```

Parameter **`method`** represents the choice of numerical integration formula, used for reactive models (e.g. capacitor, inductor, etc.) The simplest formula is Euler-backward formula (or Gear 1 formula), where time derivative in the time instant t_{n+1} (time instant to be solved) is:

$$\left.\frac{dx}{dt}\right|_{t=t_{n+1}} = \frac{x^{n+1} - x^n}{h^n} \tag{5.1}$$

where x_{n+1} is the integrated quantity in new, $(n+1)$st time instant, and x_n is already solved quantity value from nth time instant. h_n equals time step t_{n+1}-t_n.

Gear2 formula is the most popular formula for electronic circuit simulation, and is default value of parameter `method`. This is a two step formula:

$$\left.\frac{dx}{dt}\right|_{t=t_{n+1}} = \frac{2h^n + h^{n-1}}{h^n(h^n + h^{n-1})}x^{n+1} - \frac{h^n + h^{n-1}}{h^n h^{n-1}}x^n + \frac{h^n}{h^{n-1}(h^n + h^{n-1})}x^{n-1} \tag{5.2}$$

where solutions from two previous time instants, as well as two last time steps are used.

If parameter `method` equals `None`, numerical integration is not performed. That means that capacitors are treated as open circuits, and inductors as short circuits. This option is usually useful in testing circuits, to see how the circuit behave with same input, but without (parasitic) reactive elements.

Note: If `method` equals `None`, numerical integration is not performed, and the time step remains constant throughout the simulation, with the user defined value `a_step`.

If you need to have constant time step, with reactive elements taken into account, you should state in timing command:

```
a_step = a_stepmin = a_stepmax = ...;
```

Parameters `abs_LTE` and `rel_LTE` are absolute and relative tolerance of numerical integration. LTE is the local truncation error, i.e. error in one time step. For Euler-backward formula, local truncation error can be estimated as:

$$\delta = \frac{h}{2}\frac{d^2x}{dt^2} \tag{5.3}$$

where h is current time step, and the current value of the second derivative is numerically estimated. For Gear2 formula, LTE is estimated as:

$$\delta = \frac{h^2}{3}\frac{d^3x}{dt^3} \tag{5.4}$$

The tolerance ε is calculated as:

$$\varepsilon = \varepsilon_T \max\left(\text{rel_LTE}\left|\frac{dx}{dt}\right| + \text{abs_LTE}, \frac{\text{rel_LTE}\,|x|}{h} \right) \tag{5.5}$$

where expression $\text{rel_LTE}\left|\frac{dx}{dt}\right| + \text{abs_LTE}$ takes into account both relative and absolute tolerance of local truncation error. Time derivative is numerically estimated. Expression $\frac{\text{rel_LTE}\,|x|}{h}$ is the correction that takes into account numerical error in iterative process (it increases ε for small time steps *h*, otherwise error in computing can lead to further decreasing of *h*). Value of ε_T is the correction factor, which is set to 10.

Error δ is compared to ε for every reactive element (*x* is capacitor voltage, inductor current, or value whose derivative is calculated in `eqn` command). If $\delta > \varepsilon$ for at least one reactive element, the solution is discarded. (*n+1*)st time instant is calculated again, with shorter time step. We shorten the time step to set δ to be nearly equal to ε, which gives maximal time step, and the error is still within tolerances. For Euler backward formula, when calculation using eqn. (3) is used, this gives new time step as:

$$h^{n+1} = \max\left(\frac{2\varepsilon}{\frac{d^2x}{dt^2}}, \frac{h^*}{10} \right) \tag{5.6}$$

For that calculation, *x* with highest error δ is used. h^* is the discarded time step, which means, that the time step cannot be shortened more than 10 times. For Gear2 integration method, usage of eqn. (4) gives:

$$h^{n+1} = \max\left(\frac{3\varepsilon}{\sqrt{\frac{d^3x}{dt^3}}}, \frac{h^*}{10} \right) \tag{5.7}$$

If $\delta < \varepsilon$ for all reactive elements, error is smaller than the tolerance, and the solution in the current, (*n+1*)st time instant is accepted. Counter *n* is increased, and next time step is increased. For Euler backward method, this new time step is calculated using again (critical) quantity *x* with highest error δ as:

$$h^{n+1} = \min\left(\frac{2\varepsilon}{\frac{d^2x}{dt^2}}, 2h^n \right) \tag{5.8}$$

and for Gear2 method using:

$$h^{n+1} = \max\left(\frac{3\varepsilon}{\sqrt{\frac{d^3x}{dt^3}}}, 2h^n \right) \tag{5.9}$$

which means that the new time step cannot be more than two times longer than the previous.

Parameters SC_vtol, SL_itol and SDDT_tol are used for circuits with ideal switches. Alecsis posses a built-in model of ideal switch, which is unique for this simulator. It has zero resistance for closed switch and infinite resistance for open switch, and every topology of circuit is allowed. Circuits can be nonlinear, too.

For circuits with capacitors and switches, circuit should be simulated exactly at the time instant of switch transition, but before the transition occurs, to set the capacitors' voltages to correct values. After switch transition, the circuit has new topology, but the capacitors "remember" the voltages before switching . If some internal circuit voltage controls the switches, the time of switching is not known in advance, as that internal circuit voltage is obtain as the result of simulation, too. The time instant of switching is found by an iterative process. Of course, the exact time instant of switching cannot be found, so we have to introduce some tolerance. As the accuracy of capacitor voltage is in question, we have introduced SC_vtol as maximal allowed difference of capacitor voltage in two last solved time instants before the switching occurs. If all capacitors have the change of voltage below SC_vtol, we consider that we have the capacitors' voltages correct enough, i.e. we have found the switching instant correctly enough. If at least one capacitor voltage have faster change rate, solution after switch transition is discarded, time step is reduced 5 times, and the simulator searches again for the switching instant. Parameter SC_vtol has no effect if the circuit does not have inductors or ideal switches.

Parameter SL_itol is used following the same philosophy, but for the accuracy of inductor current in the moment of switching. It has no effect if the circuit does not contain both inductors and ideal switches.

Parameter SDDT_tol is used to maintain the accuracy of the quantity that is differentiated in the eqn command, if the circuit contain ideal switches. Since eqn command is used for user-defined models, the simulator does not have a clue about the order of magnitude of that quantity. Therefore, default value of parameter SDDT_tol cannot be set to some usually needed value. (For built-in components, like capacitor or inductors, order of magnitude of electrical quantities is known, so the default values can be set.) SDDT_tol has default value of 1000, which is usually too high to have any effect. It should be set by the user, according to the actual application of the model.

5.6.3.2. Control of convergence (iterative process)

Table 5.3. Control of iterative process.

Name	Default value	Meaning
vtol	1μV	Absolute tolerance for node voltage.
itol	1nA	Absolute tolerance of branch current.
qtol	1.e-20 C	Absolute tolerance of (capacitor) charge.
reltol	0.001	Relative tolerance for all variables in the system of equations.
maxiter	10	Maximal number of iterations in one time instant.
dump	0	If different from 0, iterations are dumped.
k	10	Goes with dump. Iteration dumping factor.

dcon	0	If set to 1 helps the convergence in the first time instant (t=0) by adding (temporary) conductance between every circuit node and ground. If set to 2, helps the convergence during whole transient simulation.
max_weight	1.e-4	Goes with dcon. Maximal conductance added to the main diagonal.
min_weight	1.e-12	Goes with dcon. Minimal conductance added to the main diagonal.
p	6	Goes with dcon. Parameter for calculating conductance in the next iteration. Higher value of p means that conductance value decreases faster.
q	0.5	Goes with dcon. Parameter for calculating conductance in the next iteration. Higher value of q means higher influence of current iteration number m, i.e. slower decrease of admittance value.
maxdcon	10	Goes with dcon. The maximal number of cycles.

An example is:

```
options {
        itol = 1.e-12; maxiter = 20; dcon = 2;
}
```

The first group of parameters for convergence control consists of tolerances. Alecsis checks both relative and absolute tolerances, using expression:

$$\left|x^{m+1} - x^m\right| < \left|x^m\right| \text{reltol} + tol \tag{5.10}$$

where m is iteration number, x^m and x^{m+1} are quantity values obtained in two last solved iterations, reltol is the relative tolerance (same for all quantities) and tol is the absolute tolerance. From expression (10) one can conclude that for small values of x^m, absolute tolerance is checked, and for big values of x^m, relative tolerance is checked. For node voltages, parameter tol equals vtol, for branch currents it is itol, and for charges it is qtol. For links declared as flows, tol=0, as the simulator cannot estimate order of magnitude of non-electrical quantity. That means that only relative tolerance is checked for flows.

For every link, user can define its own absolute tolerance during its declaration:

```
node [0.001] v1, v2;
flow [1.e-5] pressure;
```

For v1 and v2, tol would be equal 0.001, while for other nodes in the circuit, vtol would be used. For pressure, absolute tolerance would be 1.e-5. It is very useful to set absolute tolerances for all flows during their declaration, as relative tolerance check is not reliable for small values of x^m.

If (10) is satisfied for all links in the system, convergence is reached, and simulation can proceed to the next time step (n is increased, m is reset to 0). If (10) is not satisfied for at least one link, analysis is repeated for the next iteration. Counter m is increased, and nonlinear models are updated (calculated for new iteration).

Parameter maxiter limits the number of iteration per one time instant. It should not be a very big number. If simulator needs big number of iterations, it is very probable that the time step was too big, and the simulation result would be discarded because of local truncation error. Therefore, it makes sense to give up before reaching the convergence, and to shorten the time step. In such case, the time step is shortened 4 times in Alecsis.

If $|x^{m+1}|>10^{30}$ or $|x^{m+1}-x^{m}|>10^{30}$, Alecsis consider that there is an overflow, and the condition (10) is not checked at all. The simulator considers that also as no convergence case, and shortens the time step 4 times.

Alecsis posses two additional mechanisms to help the convergence. They are iteration dumping and node "grounding".

The user introduces iteration dumping by setting the option dump. When the dumping is introduced, starting point for the $(m+1)$st iteration is calculated as:

$$v^{m+1} \Leftarrow v^{m} + f(v^{m+1} - v^{m}) \tag{5.11}$$

where $f(x)$ is the dumping function, chosen so that $|f(x)|<|x|$. When no dumping is introduced, $f(x)=x$, which means that the starting point for $(m+1)$th iteration is v^{m+1} (solution from mth iteration). Dumping adds only part of the increment to v^{m}. In Alecsis, dumping function is implemented as:

$$f(x) = \frac{sign(x)}{k}\ln(1+k|x|) \tag{5.12}$$

This gives stronger dumping for bigger increments x. Eqn. (11) is applied on every vector member. Parameter k can be set by the user, and should be in the range (1,20).

Iteration dumping is useful if models in the system express strong nonlinearity (e.g. exponential), but it can slow convergence in other cases.

Another method of solving convergence process is node grounding. If grounding is used, conductances are connected from every node to the ground, and across PN junctions in the circuit. With high conductances, there is no doubt that the iterative process would converge easily. However, such solution is not correct, as this is not the original circuit. That is only an intermediate solution, which can be used as the starting point for solving the new circuit, where conductances are lower. In this way, conductances are decreased until their value is negligible, so we get the solution of the original circuit.

This option is activated if convergence is not obtained in maxiter iterations. Conductances are set to w=max_weight. If the solution converge, conductances are decreased in the following manner:

$$w \Leftarrow w 10^{-\frac{\mathtt{p}}{1+s+\mathtt{q}m}} \tag{5.13}$$

Conductances are multiplied by factor smaller than 1. Parameters p and q can be changed by the user. With higher value of p conductances decrease faster, while q makes bigger influence of the number of iterations m. If m is higher, conductance decrease more slowly. Paramater s is firstly set to 0. After every successful convergence, eqn. (13) is applied again, until conductances reach value min_weight, when they are considered negligible.

If convergence is not reached for some value w, conductances are set to their previous value, s is increased by 1, and eqn. (13) is applied again. With higher s, conductances decrease slower. As this process can last very long, s is limited to 10, and number of successful application of eqn (13) to maxdump (if convergence is not reached, that trial is not counted).

Mechanism of node grounding shows to be very efficient and is able to help in solving most of the problematic circuits. If dcon is set to 2, this mechanism is applied whenever number of iterations m reaches maxiter. However, you should be aware that node grounding slows down the simulation considerably. Time step shortening, which is normally performed by Alecsis when m reaches value maxiter is usually faster.

If dcon is set to 1, node grounding is applied for the initial time instant (*t*=0) only. For this time instant, convergence probems are likely to occur, as a "good guess" from the previous time point solution is not available. There are other ways to help the simulator to find the initial solution. If you set:

```
node n1=6; n3=4;
```

when you are declaring these nodes. In this way, starting values for the iterative process in *t*=0 are set. (If you declare these nodes using:

```
node n1<-6; n3<-4;
```

you can set solution for the initial time instant, i.e. circuit is not solved for *t*=0 at all - user solution is accepted.)

One can monitor conductance values using verbose level 8. If Alecsis is invoked using:

```
alec -v8 circuit_name.ac
```

Alecsis prints out information on trial number, applied conductance value, and parameter *s* value. This can be useful information for adapting values of parameters.

Note: Conductances are applied only on nodes and on PN junctions in the current version of Alecsis. If you have nonelectrical system, and unknown quantities are declared as flows, option dcon cannot help.

Note: If some convergence problem occurs, options dump and dcon should **not** be used readily. **The reason for convergence problem is very often some error in the circuit or in models.** Therefore, you should firstly check your description.

5.6.3.3. Control of system of equations solver

Table 5.4. Control of sparse matrix solver.

Name	Default value	Meaning
renum	Best (2)	Quality of sparse matrix renumeration algorithm. It can be None (0), Fast (1), or Best (2).

There is only one option that controls sparse matrix solver, and that is renum. It can change the CPU time necessary for simulation. The number of nonzero elements in the system matrix generated during LU decomposition depends on the ordering of matrix rows and columns. This reordering is performed only once, at the beginning of simulation.

If you chose option Best, a variant of Berry's algorithm is used, when very detailed (and slow) reordering is performed. This is the default value, as reordering is performed only once, and good reordering guaranties fast simulation. With option Fast, a variant of Markowitz's algorithm is used, when reordering is performed much faster, with somewhat slower simulation in time domain afterwards. This option should be chosen for very large matrices (several hundreds of equations or more), since with Berry's algorithm, reordering can take more CPU time than time-domain simulation. When option None is chose, no reordering is performed. This is implemented for comparison only, it has no practical effect, since simulation can take too much time.

Note: Values of parameter renum - None, Fast and Best are actually integer values, defined in standard Alecsis header file alec.h. In that file, it is defined:

```
#define     None      0
```

```
#define     Fast      1
#define     Best      2
```

Therefore, to use textual values of parameter `renum`, you should have file `alec.h` file included before your `root module` definition, using command:

```
# include <alec.h>.
```

5.6.3.4. Control of models

Table 5.5. Control of component models.

Name	Default value	Meaning
`charge_model`	`0`	Applicable to built-in BSIM model of MOS transistor only. It can be 0 or 1.
`temp`	`300.0`	Ambient temperature.

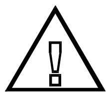
Example:

```
options { temp = 400; }
```

Option `charge_model` is applicable to BSIM model of MOS transistor that is built in Alecsis. a nonlinear capacitance can be modelled correctly only over charges. If this is not performed, problem known as *charge non-conserving* can appear. In this way, new unknowns (charges) are added to the system of equation. Nevertheless, charges can be mathematically eliminated from the system of equations, when the model is still correct but the system of equation is smaller.

If `charge_model=0`, charges related to every terminal of MOS transistor are not appearing in the system of equations. As there are four such charges associated to every MOS transistor, size of the system of equations can be much smaller. If `charge_model=1`, system of equation is bigger, but accuracy of simulation is better controlled, as the convergence is checked also for charges, using parameter `qtol`.

Option `temp` sets the value of ambient temperature. It is passed to all built-in models in the system, where model parameters are recalculated for given temperature (as in SPICE). Temperature is given in Kelvin degrees. Besides, value of this option can be used in user defined models. If keyword `temp` appear in some expression in the code, it represents the temperature value set using command `option`.



In the current version of Alecsis, option `temp` does not work for built-in (SPICE-like) models. As temperature dependence is already programmed in built-in models, we would probably improve that very soon.
For user-defined models, option `temp` works correctly, i.e. keyword `temp` used in model code returns correct temperature.

5.7. Model cards

Action parameters enable for every component to receive certain parameters that determine its behaviour in the circuit. If the number of parameters in not very big, this is an easy-to-use method. However, with the increase in the number of parameters it becomes tedious to write parameters for every component separately, especially when more components share the same parameter values. It is much more convenient group those parameters, give the group a name, and then associate that name when connecting the component. This is the concept of model cards. This concept is familiar from the simulator SPICE, and is improved in AleC++ to allow object-oriented modelling.

AleC++ supports SPICE syntax of model cards for several analogue components - MOSFET, BJT, JFET, diode. This enables usage of available SPICE model cards in Alecsis. Keyword `spice` is used to switch on SPICE model card syntax.

```
module inverter (output, input, vdd, vss) {
   mosfet mup, mdown;

   mup (output, input, vdd, vdd) { model = MNPMOS; l=2u; w=6u; }
   mdown (output, input, vss, vss) { model = MNNMOS; l=w=2u; }
}

spice {               // transition to SPICE syntax
* SPICE syntax comments
.MODEL MNPMOS PMOS ( LEVEL=1
+ VTO = -0.92V GAMMA=0.9 LAMBDA=0.1 )

.MODEL MNNMOS NMOS ( LEVEL=1
+ VTO = 0.87V GAMMA=0.67 LAMBDA=0.078 )
}                     // back to AleC++ syntax
```

Cards formed this way can be associated desired number of times when connecting components, by listing the model name after the special parameter **`model`**. The name of the model is an external symbol. Therefore, the model card itself can be given before or after referencing in the text, or can be stored in a library. As was the case with modules, when referencing the name of a model you can specify the library name to solve conflicts with double names.

```
model = ttl_lib.short_nmos;
```

5.7.1. Model cards as static objects

SPICE syntax of model cards is used for built-in components. User-defined components must also have some way of grouping parameters into model cards. SPICE syntax of model cards is not convenient here, as user-defined models can be very complex, can be even composed of submodels that have their own model cards. And there is no need to follow SPICE syntax - user-defined models are not necessarily electrical.

AleC++ model cards are created using **classes**. C++ - like class is very convenient for this purpose. If the user wants to create a new type of model card for the new type of element (created by defining a module), he or she needs to do the following:

- Supply the information about the names and types of all parameters that can appear in the model card.
- Provide a mechanism that will allocate memory for the parameters (if they are pointers) and a mechanism to set parameters to default values.

- Provide a mechanism that will test the parameter values (if they are supposed to be in some range) before the simulation, and if necessary preprocess them before the simulation.
- Provide a mechanism to free the memory of a parameter-pointer
- Make parameters of a model card, associated with a module, visible in the functional part of the module.

Most of these reminds us of classes, constructors, destructors, and the visibility rules. This makes C++ syntax of classes a natural choice for model card definition.

Creating a new model card in AleC++ is the same as creating a class. Members of a class are parameters in the card. Constructor sets their initial values (default values of model card parameters), and destructor frees the memory. As in C++, constuctor has the same name as the class, and the descrutor the same name with prefix '~'. The only difference from C++ -like class is the new special function -preprocessor. Preprocessor is necessary, since testing of the parameter value range and the preprocessing cannot be performed in the constructor. Constructor is activated when the component is declared, and it gives default parameter values. Preprocessor has to be activated when the specific model card is used for the given component, and that is performed in SPICE-like style, when connecting the component. This new method has the same name as the class, but with the prefix '>'. Preprocessor does not return any result, and it does not accept any parameter.

We use the specific model card, i.e. set of parameter values, by invoking that model card when connecting components. However, we have to associate the component model with the given model card type before, when defining the component model. This is necessary to make the parameters defined in the model card visible when defining the component model. Association of a model card and a new module is performed using the name of a class, and the operator of the access resolution '`::`'.

```
class new_diode {
        double is;              // parameter is
        double eta;             // parameter eta
    public:
        new_diode();    // constructor - cannot be inline
        ~new_diode();   // destructor - not necessary
        >new_diode();   // processor
        double evaluate_current(vd);            // diode current
};

module new_diode::ndio (plus, minus) {
    ...
    action {
        process per_moment {
            double current;
            current = this.evaluate_current(plus-minus);
            ...
        }
    }
}
```

This example illustrates a few important characteristics:

– Special methods of a model class (constructor, preprocessor, destructor) cannot be `inline` because they are called by the simulator itself.

– It is recommended for the parameters to be private. More than one component can associate the same model card, so if more of them change the parameter values, an error can be created that is very difficult to debug.

– Note the difference in access rights between the **methods** of one class and **modules** that are associated to the class. Both module and models can access parameters either directly, or using the keyword `this`, since they

have the same visibility area. However, **only methods have the right of access to all members of class**; `module ndio` from the example cannot access `private` or `protected` members. Therefore, all calculations (model equations) should be defined as `public`, so they can be accessed from the `process` code (as `evaluate_current` in the example above). Modules should use these methods, not the parameters directly. Note that all other methods, except special, can be `inline` (recommended for smaller functions).

Some users can find these limitations too restrictive. Restrictions are here to make the probability of an error smaller, and to narrow the space where you should search for an error. Nevertheless, you can declare a `module` as a `friend` of a model class. Such module has the right of access to all members.

```
class new_diode {
        ...
        friend module ndio;
};
```

If a module is to be associated to a model card, that has to be added to the module declaration. If a parent module accepts the same model card as the given module, the model card can be omitted during the declaration. In such case, the module uses the model card from the parent module. In this way the card can be passed down the model hierarchy, which can very useful for complex models.

5.7.2. Syntax of model card

Model cards are defined on the global level and are subject to external linking. The syntax of the definition is

model_card:
model *class_name :: card_name <(<arguments_for_constructor>)> {<card_body>}*

card_body:
parameter_setting
card_body parameter_setting

parameter_setting:
model_lhs = model_rhs ;

model_lhs:
<class_name::>parameter_name

model_lhs . structure_member
model_lhs [constant_expression]

model_rhs:
initializing_pahrase
model_lhs = model_rhs

We have explained how the model card type is defined. We have to explain how the particular instances of the model card (sets of parameter values) are given. The keyword `model` is used, followed by the association of the model card class and the particular model card name using the operator of resolution '`::`. (see the example below). If the constructor with arguments is used, the list of arguments in parentheses follows that association.

The body of a model card is in parentheses ('`{`' and '`}`'). The card consists of series of commands of assignment, which set the initial values of parameters. The same set of rules applies for initialization of parameters

as was the case with the initialization of static objects (it is possible to do an aggregate initialization of composites and structures). You can initialize more parameters using the same command (e.g., `a=b=c=2;`). You can also initialize only a part of a vector or a structure. If a model card is derived (i.e. class is derived), that is inherits one or more base ones, and if some parameters share the same name, you can explicitly give the name of a class and operator and the access resolution operator '`::`' to clear which parameters is to be used. If some parameters are not assigned a value in the model card, they keep their default values given in the constructor. The whole model card can be empty, when all parameters keep their default values.

```
struct S { double a, b; };

class X {
        int a;
        double b;
        char *s, v[20];
        S s1, s2;
        double m1[2][2], m2[2][2], m3[2][2];
    public:
        X (int);
        ~X ();
        >X ();
};

model X::x (2) {           // definition of model card x of type X
    a = 2; b = 5.6; s="string1"; v="string2";
    X::s1 = { 2.2, 3.5 };
    s2.a =4.7;      s2.b = 6.8;
    m1={ {1,2}, {3,4} };  //automatically converted to type double
    m2[0] = {1,1}; m2[1] = {0, 3};
    m3[0][0]= m3[0][1] = m3[1][0] = m3[1][1] = 5.6;
}

module X::M () {             // module M uses model card type X
    module X::Y y1, y2, y3; // module Y uses model card type X
    module Z z;
    module K::MK k1, k2;    // module MK uses model card type K

    y1() model = x;    // O.K. - association of model x
    y2();              // O.K. - card inherited from a parent (M)
    z() model = x;     // ERROR - Z does not accept model cards
    k1() model = x;    // ERROR - MK accepts class K, not X
    k2();   // ERROR - parent takes class X, k2 expects class K
    y3() private model = x;   // copy, not a reference of model x
}
```

Association of a model card is passing by reference - address of the model card is accessed. That means that **no copy** of the model card is created when it is associated during connection of some component. Model cards are normally only read during the simulation, parameter values are not changed. By creating copies, we would spend memory without any need. However, someone can create class methods that change the values of parameters during the simulation run, which can cause problems with other components referencing it. In that case, it is better to make a **copy** and not reference using the keyword `private` before the word `model` (see connection of component `y3` in the example above). A copy of the model card `x` would be created, and constructor and preprocessor will be applied. Whenever the keyword `private` is used, a copy of the card is created. The rest of the components will have the reference to the original one. Compiler makes a shallow copy, but can do the deep copy if you define the copy constructor - `X(X&)`. By creating private copy of the model card, we are sure that changes in a private card do not affect other components. The price for this is higher usage of memory.

Note: In this context, keyword `private` is related to creation of private copies, and is not related to the rights of access to parameters - `public`, `private` and `protected` parameters keep their access attributes unchanged.

5.8. Modules with variable structure -- clone and allocate

Modules with variable structure (application of commands **`clone`** and **`allocate`**), are explained in the Chapter on analogue simulation. It applies without any difference to digital and hybrid simulation, too.

5.9. Visibility area (name masking)

Modules create a separate visibility area. Priority rules (masking of the entities with the same name) are the following, starting from the lowest priority level:

- global objects -- the lowest priority (the widest visibility area);
- parameters and methods from model class, if a module accepts a model card;
- formal links, if any;
- local links; components connected in the structural area of the module;
- `action` parameters, if any;
- local variables in the `action` block;
- local variables in `processes`, if any;
- local variables in every new block opened inside a `process`.

The priority increases from top to bottom of the list given above, while the appropriate visibility area decreases. Every declared entity can mask an entity of the same name, which was declared in the lower priority level.

Note: Compiling with option `-O` (using optimizer) gives warning whenever masking occurs.

Masking is not a redefinition, since redefinition means for two symbols of the same name to appear in the same visibility area. Redefinition is treated as error.

6. Digital simulation in Alecsis

The previous chapter has introduced basic terms and concepts used in Alecsis for both analogue, digital and hybrid circuit simulation. We are now going to focus on each group separately. This chapter discusses digital simulation in Alesis.

6.1. Alecsis support for digital simulation

Logic simulation can be performed using different systems of logic states. For each system of states, new logic gates have to be developed. We generalize this as a problem of discrete-event simulation, which is not necessarily electrical. Therefore, it is very useful if the user can create his own system of logic states, as well as logic operators (gates).

Alecsis was designed as an offspring of C/C++. These languages do not have too many specialized constructs, and can wok up to their full potential only in conjunction with various libraries, which cover numerous applications. In a similar manner, Alecsis does not have built-in digital component, or a formal system of states, or number of logic states. It provides only a discrete-event simulation engine, i.e. a mechanism that can process events and queue new events that come as results. If libraries of logic states and basic logic gates are not available this can be serious drawback. However if a library supporting different styles and types of digital simulation is available, this becomes a major advantage. User of already prepared libraries can consider Alecsis as an ordinary logic simulator. But and advanced user can create his own libraries and adapt Alecsis for his problems.

If we want to use Alecsis with a new system of logic states, we have firstly to prepare *program support*. That program support is used later in creation of appropriate logic gates. There are three steps in creation of the program support:

- creating the system of states (new enumeration type),

- creating new logic truth tables,
- overloading of appropriate logic operators, to accept links of the new type
- if model cards are to be used, declarations of model class and definition of methods are necessary.

In this way basis for simulation with new logic system is created, which can be used for creation of models.

6.1.1. Systems of states

System of state of a simulator describes legal states of a digital simulator. Total number of these states is called the cardinal number of the simulator. Many simulators have a built-in system of state. Alecsis does not have such a system, and it needs to be defined before the simulation (often used systems are available in the library).

You need to use enumeration types to define the system of state. To allow for representations of digital words (vectors), system of state needs to be defined with characters as symbols. Examples are:

```
typedef enum { 'x', '0', '1' } three_t;      //system with 3 states
typedef enum { 'x', '0', '1', 'z' } four_t; //system with 4 states
typedef enum { ' '=void, '_'=void,
               'x', 'X'='x', '0', '1', 'z', 'Z'='z'} four_t;
```

The last example defines a type with 4 states, two types of separators for enumeration strings, and is case-insensitive. Enumeration types in AleC++ occupy 1 byte (char), taking that the indexes of symbols are in range -128 to 127, otherwise they take 2 bytes.

Term "system of states" is used conditionally, since the construct `typedef` merely defines a new name for a enumeration set. Thus, an unlimited number of different "systems of states" can exist on the global level simultaneously, taking the names are different.

The order of symbols in the set is important for three reasons:

- If sets are non-initialized, values are given as in C/C++ - starting from value 0 for the first signal.
- Order of states is important for creating truth tables of logic functions;
- Signals (digital links) that are not initialized, have the starting value that is the first symbol in the set. In our example, all signals in the circuit of type `three_t` and `four_t` that are not initialized will have starting state `'x'`, since `'x'` is the first symbol in the set (separators do not count). Therefore, undefined state `'x'` should be always the first symbol, as it is usual to treat non-initalized signal as undefined when the simulation begins.

6.1.2. Truth tables

AleC++ has basic binary logic operators needed for logical simulation. However, they accept integer operands, not enumeration ones. This means that for every new system of state we have to define truth tables for all operators. Truth tables for unary operators are vectors of length N, for binary -- matrices NxN, where N is the number of states in the actual system of states (the cardinal number).

```
const three_t nottab[] =      { 'x', '1', '0' };
                            //   x    0    1

const three_t andtab[][3] = { { 'x', '0', 'x' },   // x
                              { '0', '0', '0' },   // 0
```

```
        { 'x', '0', '1' }    // 1
    };
```

6.1.3. Overloading of operators

This tables alone are sufficient, and can directly be used for modelling of a digital circuit. It may be better, however to use operator overload:

```
inline three_t operator~ (three_t op) { return nottab[op]; }
inline three_t operator& (three_t op1, three_t op2)
                                  { return andtab[op1][op2]; }
foo () {
    three_t x = '1', y = '0', z = '1', r;

    r = ~(x & y & ~z);
}
```

Overloaded operators are very simple functions, which use truth tables, and return whatever was on a particular position. Being so short, they are suitable to be `inline`. It is usual for that definition of an `inline` operator to be in the same file with the definition of the system of states. That file can be included as a header file into a new file, where circuit description is, using `include` command:

```
# include <declaration_file.h>
```

Table declaration can be in the same file (as `extern`), but the definition (with the initialization) ought to be in a separate file. That may be a file storing all definitions for a particular system of states, and can be compiled into a library (using `-c` option), by invoking:

```
alec -c definition_file.ac
```

That library can be used later by Alecsis linker. If the user prefers common, and not inline operators, than the declaration of the operators remains in header file, and their definitions are compiled together with table definitions. As you can see, the intention is use C/C++ programming style - to group the declaration into one header file that can be included when necessary, while the definitions are compiled and stored as library.

Assignment operators can be also overloaded. It stores the result of a logical operation into the left operand (&=). The left operator need to be passed by reference to the operator function:

```
inline three_t operator &= (three_t& op1, three_t op2)
                           { return (op1 = andtab[op1][op2]); }
```

6.1.4. Overloading operators for vectors

All logic operators can be overloaded for a new system of state using appropriate tables. Nevertheless, if you want simulation on RT level (RTL -- register transfer level), you need to have operators for vector-operands overloaded, too. To make that possible, operator function needs to posses information about the vector dimension. A mechanism that is used for strings (`strlen` function) cannot be used. Symbols in systems of state can have a value of `'0'`, which renders the mechanism of classical strings (with `'\0'` as an *n+1* element) inoperable. For that reason, AleC++ has a special operator **`lengthof`**, which does not exist in C/C++.

6.1.4.1. Operator lengthof

This operator is syntactically very similar to the operator `sizeof`. It is a unary operator whose **operand needs to be a vector of enumeration type**. (vector of signals or vector of variables). The vector can be a formal variable, local variable, or a formal/local signal. In the case of a formal variable, AleC++ passes as a hidden parameter the length of the actual argument extracted by the operator. The situation is much simpler with signals -- a special instruction of the Alecsis virtual processor that returns the length of the signal-vector. Note that operand of `lengthof` operator cannot be a local pointer, because `lengthof` would return 0 in that case.

If a vector is declared with inverse dimensionality, the `lengthof` returns the **negative value** of its length. Usually, for overloading vector-operands, you need the absolute value of the length. Direction can have an important role with some specific operators (left shifting << or right shifting >>).

```
three_t v1[0:3], v2[3:0];
int L1 = lengthof (v1);   // L1 is 4
int L2 = lengthof (v2);   // L2 is -4
```

6.1.4.2. Buffers for temporary solutions

Logical operators often return a result, that can itself be an operand in a more complex expression. Since we are dealing with a vector, a temporary memory location needs to be reserved for the result, and another temporary memory to prevent the overwrite of that result by the new operation. The following example illustrates this:

```
# define Vreturn(v,size) {asm movq.l %d4, size; return v; }

# define BUFF_SIZE 512
three_t _op_buff[BUFF_SIZE], _res_buff[BUFF_SIZE];

three_t *operator& (three_t *op1, three_t *op2) {
    int size1 = abs (lengthof op1);
    int size2 = abs (lengthof op2);
    if (size1 != size2) {
      printf("incompatible lengths - op &, type three_t\n");
      exit(1);
    }
    for (int i=0; i< size1; i++) _op_buff[i] = op1[i] & op2[i];
    memcpy(_res_buff, _op_buff, size1);
    Vreturn(_res_buff, size1)
}

three_t *operator= (three_t *op1, three_t *op2) {
    int size = abs(lengthof op1);
    memcpy(op1, op2, size);
    memcpy(_res_buff, op1, size);
    Vreturn(_res_buff, size)
}

foo () {
    static three_t v1[]="0100", v2[]="11x0", v3[4], v4[4];
    v4 = v3 = v1 & v2;
}
```

The example illustrates a concept of using two buffers -- one for calculations, the other for the return of the result. In this particular example use of only one buffer would be a satisfactory solution. However, in some more complex operations it may happen for one of the operands to be a result of an overload operation, which may mean that we are reading and changing the same vector simultaneously. That may produce errors that are very difficult to

debug. It is better to use two buffers - the result is formed in `_op_buff` buffer, and is then copied into `_res_buff` buffer.

The return of a vector is rather uncommon -- we have used our macro `Vreturn`. Macro `Vreturn` returns not only the pointer to a vector, but also its length, which enables later usage of command `lengthof`. From `Vreturn`, usual command `return` is invoked as:

```
return v;
```

where pointer `v` is returned. Pointer is stored into lower register of the virtual processor accumulator %d0. But before invoking usual command `return`, we have stored the length of the vector into %d4 (the higher accumulator register). Assembler command is used for that. If this result is passed to a function as an operator, operator `lengthof` can determine the vector length from this higher register. Macro `Vreturn` is available in the standard header file "`alec.h`").

With a similar overload of operator =, you can handle enumeration vectors the same way as ordinary numbers. Therefore, the user of the logic simulator, which has a library already prepared, can be unaware of both `lengthof` operator and temporary buffers, as in function `foo` in the example above.

It is up to the designer if he or she is going to allow for the combination of vectors of different length and/or different direction in such binary operators.

Unary operators of left and right shift (<< and >>) can be overloaded in this manner, as well as increment, and decrement (++ and --), etc.

6.2. Synchronization of digital processes

Such program support, consisting of prepared systems of states, logic truth tables, and appropriate operators is now available to model digital components. On the other hand, we have available an internal *Alecsis discrete-event selective-trace* simulation engine, that is able to manage the logic events. The construct *discrete-event* means that the simulator does not solve the time-continuum, but only time instants when a logic event (change of logic state) happens. Outside of these time-instants, there are no changes in the circuit, i.e. nothing for the simulator to solve. *Selective-trace* means that only some models are executed when an event happens. When an event is generated at the output signal of some digital component, it activates only those components where the same signal acts as input. Therefore, there is no need to simulate the whole circuit, only activated components (models) are simulated. Therefore, we can consider execution of logic simulation as execution of *synchronized processes*, and *transmission of messages via signals*.

When we are defining digital models, we are using the program support from the library. We have to provide the information for the simulation engine how to synchronize the processes. Here is an example:

```
module and2 (three_t in a, b; three_t out y) {
    action (double delay) {
        process (a, b) {  y <- a & b after delay;  }
    }
}
```

We have modelled logic AND circuit with two inputs for system of states `three_t`. This `module` does not have structural constructs, i. e. we are not using previously defined components as submodels of this model. The functional part consists of one `process` sensitive to events at the inputs -- signals `a` and `b`. Whenever an event happens to either one of them, the `process` activates, and executes the signal assignment command (operator `<-`). We have used logic operator &, overloaded to accept signals of type `three_t`. The result will become the new value of signal `y`, but not immediately. A delay of `delay` seconds is introduced. Since that is the last command of the process, the control is transferred to the beginning, and the process is suspended until the next event on some of the inputs.

6.2.1. Interpretation of signals in expressions

The name of a signal can legally appear in the expressions of the process. (Be aware about the masking rules given in the previous chapter.) However, they must be treated differently than C/C++-like variables. Links, and therefore signals as digital links, are complex entities, which can be differently treated in different situations**. The treatment of the signal depends on the context.**

In all **non- assignment expressions**, the name of the signal refers to **the memory storing the current value of the signal**. The size, dimensionality, and the type of the memory depend upon the type of the signal (scalars, vector, or a structure). In other words, in such expressions, the name of the signal refers to it value - it is used as a variable.

Signals must not appear on the left side of the assignment operator (such as =, =+, ++, etc.) since they are not l-values. The change of the signal value can be performed exclusively using a special AleC++ assignment operator for signals `<-`. (Note that symbol `<-` is used differently for analogue links, where it can be used in analogue link declaration to set its value for the first time instant (*t*=0).) The simulation engine is responsible for assigning the value of the signals, as they are quantities that appear in the circuit connections. This information is given to the simulator by the use of the assignment operator `<-`. You cannot just write into signal.

The rules addressing the agreement of types, access to the members of structures, indexing of vectors, etc. are the same as for variables.

```
typedef enum { 'x', '0', '1' } three_t;
struct S { three_t send, recv; };
module X (three_t a, b[], c[][10]; S s1, s2[]) {
   action {
      process (a,b) {
         a;          // value of signal a - scalar
         b;          // pointer to position b[0] of vector b
         b[2];       // scalar  from position 2 of vector b
         c;          // matrix c (pointer to c[0][0])
         c[2];       // vector of length 10, position 2, matrix c
         c[2][3];    // scalar, position 2,3 matrix c
         s1;         // structure, type S
         s1.send;    // scalar - element of structure s1
         s2;         // vector of structures
         s2[1];      // structure at position 1, type S
         s2[1].send;//scalar - element of structure at position 1
      }
   }
}
```

Formal signals that are arrays do not need to have a specified first dimension (as `b[]`, `c[][10]`, and `s2[]` in the example above) since that dimension is not necessary for indexing (operator `lengthof` will determine it with enumerated variables) **Nevertheless, all other dimensions are necessary for arrays with more than one dimension.** Formal signals do not exist independently in the memory. Only global or local entities exist as independent entities in the memory. Formal signal is just the name used for matching the appropriate pair on the actual side.

Signals can be scalars, if composite homogeneous (vectors, matrices, etc.), and heterogeneous (structures), but they are also internally represented as collections of scalars. For that reason is the `process` that is sensitive on a signal-vector, actually sensitive to every scalar making the signal-vector.

6.2.2. Conversion of link type

If a link has more than one aspect, i.e. it is both digital and analogue, the **operator of link type conversion** can solve the ambiguity. This operator is very similar to C/C++ **cast** operator used for type conversion.

```
signal s;
action {
    process {
        double s_as_node;
        s_as_node = (node)s;
    }
}
```

We can perform link type conversion by stating (in parentheses) the desired type before the signal name. Priority and the associating rules of this conversion operator is the same as was with cast operator.

Note: A wrong assumption that signal `s` possesses analogue aspect will result in a usually harmless value of 0 during the simulation. However, if we try the reverse, to convert the node into a signal, where the node does not have digital aspect, effect can be harmful.

6.2.3. Conditional process suspension -- command wait

We have already said in the previous chapter that a `process` can be synhronized by giving the list of signals it is sensitive to. Another method of synchronizing a `process` is using command **`wait`**. Command `wait` suspends the current `process` for a period of time, giving t the same time conditions for its reactivation.

wait_command:

wait <*sensitivity_list*> <**while** *suspension_condition*> <**for** *time-out* >*;*

An empty command is also legal:

```
wait;
```

In this case, the `process` is suspended until the end of the simulation. The *sensitivity_list* is the same list of signals that can be defined in the `process` heading (the `process` using `wait` command must not be synchronized using some other method). The `process` execution stops at the `wait` command, until an event happens to some of the signals the command is sensitive to. If *suspension_condition* is given, it has to become zero, if the `process` is to activate, regardless of the events on some of the signals from the *sensitivity_list*. If *time_out* is defined, all of that can last up to *time_out* seconds.

```
enum Edge { RisingEdge, FallingEdge };

module (three_t in clock) {
    action (Edge mode) {
        process {
            ...
            switch (mode) {
                case RisingEdge:
                    wait clock while clock != '1' for 100ns;
                    break;
                case FallingEdge:
                    wait clock while clock != '0';
                    break;
            }
```

```
                // code activates at the edge of clock signal
                ...
            }
        }
    }
```

In this example, `wait` command is sensitive to the clock signal, with the condition of a transition to `'1'` (if `action` parameter `mode` equals `RisingEdge`) or `'0'` (if `mode` equals `FallingEdge`). The appropriate edge of the clock activates the code that follows (omitted in the example above). At the end of the process, the control is back at its beginning, and the execution stops again at `wait` command, waiting the appropriate clock edge.

6.2.4. Predefined signal attributes

All signals, regardless of their data type, have a number of attributes, whose value can be accessed in a `process`. All attributes are of `int` type, and can be accessed using indirection operator `->` (signal name is treated as a pointer, i.e. address in memory). Signal attributes are:

- **`active`** -- 1 if the signal is active in the present moment, otherwise 0
- **`event`** -- 1 if an event is currently happening on the signal, otherwise 0
- **`quiet`** -- 1 if the signal is not processed in the present moment (`!active`), otherwise 0
- **`stable`** --1 if the signal is not changing value (`!event`), otherwise 0
- **`fanin`** -- number of processes sensitive to this signal
- **`fanout`** -- number of drivers of this signal
- **`hybrid`** -- 1 if the signal has also analogue aspect, otherwise 0

It can happen that the signal is processed, but the result is the same value as it was before. Processed signal is `active`, while an `event` happened to it only when the new value is different from the old one. The term sensitivity has already been explained, while the concept of drivers will be explained in the following section. The last attribute indicates the `hybrid` aspect of the present signal, that is the coupling of analogue elements (which can have effect on delay calculations and other activities).

Note: The indirection operator `->` can be used without restrictions, since the members of signals-structures can be accessed using character `'.'`, and **the signal cannot be declared as a pointer**.

Composite signals have attributes `active`, `event`, `quiet`, and `stable`. Attributes `hybrid`, `fanin`, and `fanout` are not allowed it the signal is not a scalar. Attributes `active` and `event` are obtained as a result of *or* operation on attributes of all scalars of the signal (for example, a vector is `active` if at least one position is `active`). Attribute `quiet` is negation of `active`, and `stable` is negation of `event`.

```
signal four_t v[4];

...
v[0]->fanin;  // o.k.
v->fanin;     // error - composite signals cannot that attribute
```

We use attributes in the following example for detection of static hazard - simultaneous and opposite change of the inputs of an AND circuit).

```
module and2 (three_t in a,b; three_t out y) {
   action (double delay) {
      process (a,b) {
         if (a->event && b->event && a != b)
            warning ("static hazard at inputs of AND2 circuit");
         y <- a & b after delay;
      }
   }
}
```

6.2.5. Signal assignment -- operator '<-'

The change in the signal value is possible only using operator `<-`. The difference between this, and an ordinary assignment of some value to a variable, is in that the ordinary assignment happens instantaneously, that is the next reference to the variable gives the new value. Assignment of signals is not performed instantaneously, it is information for the simulation engine to give a new value to the signal. It has a certain delay, so that the new value cannot appear before the next cycle of simulation. Such assignment can be delayed so the new value appears after a certain period of time expires. The delay models the inertia of the signal through a physical component in digital simulation (in analogue simulation, delay occurs indirectly by solving differential equations in transient simulation). In idealized case, when the delay is 0, the assigment is also not performed instantaneously, but in the next delta-cycle, i.e. next time when the simulator treats the events that are waiting in the queue to be processed.

The concept connected to signal assignment in AleC++ comes from a language for description of digital circuits VHDL. In the following section we describe properties of operator `<-` and its effects.

6.2.5.1. Drivers

When compiler reaches `<-` operator, it does not know whether some other process assigns the value to the same signal. That signal can be passed to a module using interface (formal signal), or can be a local signal that is passed to some submodule, so the simulator is not aware about all connections to that signal when executing the assignment statement. This technique of connecting is used for production of digital circuits known as WIRED-AND and WIRED-OUT, when outputs of more digital circuits are connected to a bus. In this way, logic function can be executed with savings in hardware and with reduced delay.

When it happens that more statements assign the value to the same signal, a resolution function is invoked. To enable this, assignment is never done directly, but using intermediaries -- drivers. **Every process creates a driver for that signal. Assigning into the same signal in two different processes creates two drivers. Signal with multiple drivers has to have a defined resolution function**, which will derive the final value of the signal from the combined values from drivers. Since a **driver is characteristics of a scalar signal**, the assignment to a signal-vector creates drivers for each particular scalar index.

```
signal three_t s1, s2[4], s3[2][4];
..
  process (...) {
    int i, j;
    three_t m[2][4] = { "1101", "0101" };

    s1 <- 'x';  // creates 1 driver for s1
    s2 <- "0100";       // 4 drivers - from s2[0] to s2[3]
    s2[3] <- '2';       // 1 driver for s2[3]
    s2[i] <- '2';       // 4 drivers - i is a constant
    s3 <- m;            // 2x4=8 drivers for s3
    s3[1] <- "0101";    // 4 drivers - from s3[1][0] to s2[1][3]
    s3[i] <- "0010";    // 8 drivers - i is not a constant
```

```
    s3[1][0] <- '0';     // 1 driver for s3[1][0]
    s3[1][i] <- '0';     // 4 drivers - i is not a constant
    s3[i][j] <- '0'; // 8 drivers - both i and j are not constants
  }
```

Drivers are created during compilation, and their number has to be known before the start of the simulation.

Signals on the right-hand side of `<-` operator can be composite, but not heterogeneous. During the type checking, compiler performs implicit conversion, as for all expressions.

6.2.5.2. Delay models

The previous assignment examples did not have a defined delay, which means that the new value will appear on the signals in the next cycles of the process (although the time indicator `now` is set to the same time-instant in all that cycles). This delay is a consequence of the simulation algorithm, which gathers the assignments, and then processes them. Nevertheless, assignment operator can be used with user-defined delay. This delays the processing for a specific period of time.

signal_assignment:
signal **<-** <**transport**> *new_value ;*

new_value:
expression <**after** *expression*>
list_of_new_values

list_of_new_values:
pair_expression_time
list_of_new_values , pair_expression_time

pair_expression_time:
expression **after** *expression*

A signal can be assigned a new value with or without the key word **`transport`**, which concerns the type of delay. If it is used, the delay that is applied is characteristic for transmission lines -- every impulse is transmitted, regardless of its length. If the keyword `transport` is not used, the delay is **inertial** -- impulses must last long enough to produce an effect. Interial delay is characteristic for digital components, where impulse must have enough energy to change the state of the digital component.

The expression after the keyword **`after`** regards the assignment delay with respect to the present time instant. Signal can accept a sequence of pairs *value-time*, as well.

New values of signals and their delays represent future events, which are stored on the internal list of every driver. These future events are ordered in the list by their time - ordering of ascending time. However, there are some rules of arrangement that depend upon the type of delay used:

- In case of **transport** delay, the new pair *value-time*, that is new events will be put on the list so that all pairs *value-time* whose *time* is greater than the time of the new pair will be erased;
- In case if **inertial** delay, all pairs on the list whose *time* is less than the *time* of the new pair will be erased. The exceptions are the pairs whose *value* is the same with the *value* of the new pair.

This type of arrangement of future events agrees with the behaviour of real digital components. The event caused by a short-lived impulse will be erased in the case if inertial delay before its time comes, so the component that owns the driver will not react to the impulse.

Simulator constantly controls the time of the first event on the list of every driver. If the time agrees with the present moment, the value of that event becomes the new value of the driver. In order for the new value of driver to become the new value of the signal, at least one of the two conditions needs to be met:

- Signal has only one driver;
- Signal has an appended resolution function, which will resolve the final value from driver values.

6.2.5.3. Resolution of conflicts on the bus (resolution function)

A signal with two drivers gives rise to **wired logic**. We can approximate a number of outputs of digital circuits using voltage sources with their output resistances. By superimposing these sources, we can arrive at the resulting value at a node. If the output resistances differ significantly, the resulting value in a node is the result of the source with the smallest output resistance. Speaking in wired-logic terms, signals can, beside the state, have **intensity**. The following conditions need to be met if the simulator is to model wired logic:

- System of states has to have a state of high impedance, indicating that the driver is currently off-line;
- Every multiple-driver signal has to have a resolution function.

```
typedef enum { 'x', '0', '1', 'z' } four_t;

four_t bus4res (const four_t *drv, int *report) {
    int ndrivers = lengthof drv;
    four_t result = drv[0];
    for (int i=1; i<ndrivers; i++) {
        switch (drv[i]) {
           case '0':
           case '1':
              result=(result=='z'||result==drv[i])?drv[i]:'x';
              break;
           case 'x':
              result = 'x';
              break;
           case 'z':
              break;
        }
    }
    if (ndrivers > 1 && result == 'x') *report = 1;
    return result;
}

signal four_t:bus4res bus[3:0];
signal four_t:bus4res line = '0';
```

Function `bus4res` given above makes the resolution for the system of four states. If at least only one driver is `'x'` (undefined), the result is also `'x'`. If none of the drivers is `'x'`, but there are at least two of them having different logic values `'0'` and `'1'`, the result is again `'x'`, as there are two drivers with different values and high intensity. The wired logic is normally used when only one driver drives the bus, which means that all others are at high impedance state `'z'`, that is, have weak intensity.

The resolution function is a global function fulfilling certain conditions regarding results and parameters. If a signal is of *link (signal) data type* `T`, its resolution function will return the same type. The function has to have two parameters, a pointer to data type `T`, and a pointer to `int` type. The former is a vector of instantaneous values of all drivers of a signal, and the second is a flag indicator. This flags conveys whether the resolution needs to report a conflict on the bus or not (the flag needs to be set to 1 for the message to appear).

An association of a signal and its resolution function is obtained on signal declaration, if the character `':'` and the name of the resolution function are listed after the signal data type. Another way is to create new type (using `typedef`) that includes the resolution function:

```
four_t bus4or  (const four_t *drv, int *report);
four_t bus4and (const four_t *drv, int *report);

typedef four_t:bus4or four_t_or;   // new, resolved type fout_t_or
typedef four_t:bus4and four_t_and; // second version

typedef struct {
    four_t send, recv;
    four_t:bus4and data[3:0];
}:bus4or Connector;
```

The last example shows the application of the resolution function in case of structures. The whole structure has a resolution function `bus4or`, which is applied on members `send` and `recv`. However, structure member `data` has its own resolution function `bus4and`.

For one system of states, you can define a desired number of resolution functions, e.g wired-AND, wired-OR. For system of states with high number of states it can be useful to create tables of resolutions as arrays.

If a signal has one driver, and its resolution function is defined, it is applied. However, signal with one driver needs not a resolution function. In such case, driver value is copied to the signal.

6.2.5.4. Driver initialization

If the system of states begins with `'x'` (the first enumerated symbol is indexed as 0), all non-initialized signals in the circuit would be set to `'x'` at the beginning of the simulation. For most of the cases, this is natural choice, as `'x'` denotes undefined state. Nevertheless, for modelling of switching logic this can be a problem. Transmission gate (switch) is actually a MOS transistor, and drivers for *source* and *drain* would be set to `'x'` even if the gate is open. Such transmission gates are usually used for connections to the bus. Undefined state `'x'` on any driver of the bus means usually that the state `'x'` would be resolved for the whole bus. However, correct model of real circuit behaviour demands that an open switch means that bus is driven with high impedance state, `'z'`.

The solution is found by initialization of formal signals.

```
module tgate (four_t in gate; four_t inout drain='z', source='z');
```

If formal signals have direction `in` or `inout`, and an initial value is given, then all drivers created by the `processes` of that `module` will have that initial value.

The problem can arise with initialization of multidimensional-array-signals. Their first dimension can be unknown, as they are formal signals, so one cannot utilize aggregate initial array of values with fixed length. In such cases, one can use operator `<-`, instead of `=`, which means that the given initial value is repeated to cover the whole length of the signal:

```
module X (four_t out y[] <- "z");
```

In the above example, all drivers created by `processes` of `module X`, for all indexes of vector `y`, will be in `'z'` state, whatever the size of the actual signal that corresponds to the formal signal `y` is.

6.2.5.5. Complex delay

There are many ways of modelling delay in logic simulators (unified, assigned, rise-fall, static, etc.). When the delay is not a predefined value, the best way is to express it as a result of a function. If the function is not too long, you can define it as `inline`:

```
double fdelay (three_t new_value, double tplh, double tphl) {
    if (new_value=='0') return tphl;
    if (new_value=='1') return tplh;
    return tplh > tphl ? tplh : tphl;
}

...
process (a,b) {
    three_t res = a & b;
    y <- res after fdelay (res, 10ns, 12ns);
}
```

When dealing with complex models of delay, such as minimum, typical, and maximum values, we recommend gathering of all parameters in model card. You can then define the functions that calculate the delay value as methods of model classes. Of course, other functions that are used for component modelling are suitable to be methods of model classes, too.

6.3. User-defined signal attribute -- operator '@'

Beside seven signal attributes generated and controlled by the simulator itself (`active`, `event`, `quiet`, `stable`, `hybrid`, `fanin`, and `fanout`) there is an user-defined attribute, too. Memory for this attribute is separately allocated. The name of the signal refers to its current value, while the name of the signal preceded by the operator '@' denotes the pointer to user-defined attribute. User-defined attribute is allowed for scalars signals only (including members of arrays or structure members).

Attribute can be of any legal type, including classes, but cannot be a vector, reference, or a pointer. Since one can declare a class as an attribute, there can be effectively more user-defined attributes (class members). Character '@' is used for attribute declaration, too:

```
signal three_t@int s, v[4];        // attribute of type int
...
process {
    s;           // value of signal s
    @s;          // pointer to an attribute of signal s
    s=1;         // wrong - s is a signal
    *@s=1;       // O.K. - recording int an attribute of signal s
    @v;          // wrong, v is a signal-vector
    @(v[1])      // O.K. - v[1] is a scalar signal
}
```

It is not legal to use operator '@' for signal without attributes. Attributes can be incorporated into data type using `typedef` command. Such data type can be later be used exclusively for signals.

Note: One can use attributes to pass information between processes. However, this cannot be a correct model of real circuit behaviour. Therefore, **attributes should not be used as another way of communication between processes**. You should set their values in the preparation phase of the simulation, and later only use these values.

Application of user-defined attributes will be illustrated on the problem of capacitance modelling in the digital circuit. Delay of a digital circuit depends linearly (as the first approximation) on the capacitance connected to its output. Capacitances in a digital system are input/output (terminal) capacitances of the circuit, and the parasitic capacitances of connections (links). The former can be considered as module parameters, and passed as such to modules. The later need to be connected to signals. This makes connection capacitance an ideal candidate for a signal attribute. These capacitances cannot be changed during the simulation, which means we should declare an attribute of as signal as a class, whose private member is a total capacitance of the signal. The capacitance value will come out in the preparatory phase of the simulation. During the simulation run, it will be used for calculation of the module delay.

```
typedef enum { 'x', '0', '1', 'z' } four_t;

class four_att {
        double Tcap;    // total capacitance of the signal
    public:
        four_att (double cap=0.0) { Tcap = cap; }  // constructor
        void add_tcap(double cap) { Tcap += cap; }
        double tcap () { return Tcap; }
};

typedef four_t @ four_att four_full;
```

The memory is allocated for local signals only -- formal signals just denote connections to other signals, which are declared as local in some modules that are on higher hierarchy levels. Therefore, constructor for attributes is activated with arguments that are given in local signal declaration.

```
four_t @ four_att(0.2p) bus;
four_full(0.1p) line[3:0];
```

This way of declaration gives certain capacitance to the signal (parasitic capacitance). The best moment for total capacitance calculation is after forming of the hierarchical tree. Therefore, process synchronized as `post_structural` should be used:

```
module and2 (four_full a, b; four_full out y) {
    action (double Cin, double delay, double skew) {
        process post_structural {
            (@a)->add_tcap (Cin);
            (@b)->add_tcap (Cin);
        }
        process (a, b) {
            y <- a & b after delay + skew * (@y)->tcap();
            ...
        }
    }
}
```

In every component of type `and2`, signals `a` and `b` would have input capacitance `Cin` (passed as `action` parameter) added. In reality, formal signals `a` and `b` are just connections (other names) to some actual signals elsewhere, that have that input capacitance added. Parameter `skew` is the coefficient of increase of total delay with the increase of capacitance load at the output. Therefore, multiplying value `skew` with the total capacitance of the signal `y`, we get the delay added to the parameter `delay` (parameter `delay` is the delay when the output is not loaded with capacitances). Therefore, the total logic circuit's delay depends on `fanin` (the number of modules linked to the output). We cannot access the value of capacitance, since it is declared as `private`. By the use of an

additional mechanism, we can ban the use of `add_tcap` method, and thus fix terminal capacitance to be read-only.

6.4. Leaving out actual signals -- void

Some formal signals can appear to be unnecessary during connecting. For example, one can use only non-inverting output of a flip-flop in the circuit, while the inverting output is not used -- it remains unconnected. In AleC++, this is enabled by inserting the word **`void`** instead the actual signal, when such component is connected. Compiler will generate an implicit signal whose dimensionality and initial value will be determined from the formal declaration. If the formal signal has direction `in`, processes sensitive to it will never activate (if we do not count one pass during the initialization). Signals with direction `inout` can still be active if they have drivers inside the `module` that is to be connected.

```
module rsff(three_t in reset='1', set='0';
                      three_t out q='0', qbar='1') rsff1;
signal three_t r='1', s='0', q;

ruffs (r, s, q, void);
```

The implicit signal, generated automatically as a consequence of omission of actual signal, cannot be referenced to in the printout control (command `plot`). Its activity are known only to processes on the formal side of the interface.

6.5. Variable number of formal signals -- operators '$' and '$$'

In C/C++, one can declare and define functions with a variable number of arguments. The number of formal parameters of a `module` can be variable, too:

```
module andx (three_t out y; three_t in ...) {
    action (double delay) {
        process structural {
            if ($$ < 2) warning("module andx has no inputs",1):
        }
        process ($2 ...) {
            three_t result = $2;
            int i;
            for (i=3; i<=$$; i++)  result &= $i;
            y <- result after delay;
        }
    }
}
```

The last example has modelled logic AND circuit with N ($N \geq 1$) inputs. The output to the circuit is declared first, while the inputs are given by type and direction, with `'...'` instead the name. This symbol is very important for compiler, and cannot be omitted in the declaration of such module. The compiler will not check the number and type of actual signals to the right of signal `y`, if this character is missing.

Operator '`$$`' does not demand operands and returns the total number of formal signals. To get the number of inputs, we need to subtract the number of fixed signals (1 in this case -- signal `y`) from the value returned by '`$$`'.

The main `process` is synchronized to be sensitive to all formal signals to the right of signal `y`. Symbol '`$2`' means the 'second formal signal by order.' The same signal appears in the code of the `process`, as well. In the `process` code it is legal to place an expression to the right of '`$`'. The result of the operation `$expression` is a formal signal at that position, with the type and direction from the declaration in the heading.

```
module andx (three_t out y; three_t in ...) action (double delay);

module X () {
    andx and1, and2, and3;
    signal three_t s1, s2, s3, s4, s5, s6;

    and1 (s3, s1, s2) delay = 10ns;
    and2 (s4, s1, s2, s3)  delay = 12ns;
    and3 (s6, s1, s2, s3, s4, s5)  delay=9.7ns;
}
```

Operator '`$`' has the same priority as all other unary operators, which mean that parentheses must be used when dealing with a complex expression.

```
$i + 1       // signal from position, added with 1 -- an error?
$(i + 1)     // signal from position i+1
```

6.6. Variable number of action parameters

The number of parameters in the header of the `action` block of the `module` can be variable, too. This is described in the Chapter 5, as it is the same for digital and analogue modules.

6.7. Array of components -- commands clone and allocate

In Alecsis, one can generate an array of previously declared components. Commands **`clone`** and **`allocate`** is used for that. Details of command `clone` are given in the chapter on analogue simulation, and are valid for digital simulation, too.

6.8. Structural aspect of digital circuits

The final discussion in this chapter concerns the structural approach to digital simulation. This was the only way with older digital simulators, since all digital components along with tables and system of states were built-in and fixed. Alecsis does not have any built-in digital component, but it does have mechanisms for creation of extensive libraries of components. Such components can be used as they are built-in.

Digital elements use the same syntax, parameter setting, declarations, etc., as analogue or hybrid components. AleC++ does not have a construct that would allow the compiler to guess if a component of some `module` type is analogue, digital, or hybrid.

7. Analogue simulation in Alecsis

In time-domain analogue simulation, system of equations that describes the analogue circuit (system) is composed and solved in many time instants. If the circuit is nonlinear, the linearized system is created and solved in many iterations in every time instants, until the convergence occurs. Therefore, there are two loops - time-domain (outer) loop and iterative (inner) loop. The control of time-step size and convergence control are described in Chapter 5. In this chapter, built-in models and model description in AleC++ will be described.

As other circuit simulators, Alecsis is forming system of equations component by component, not equation by equation. That means, contribution of each component is determined as a "stamp", that is added to the system of equations. Therefore, discretization of differential equations, and linearization of nonlinear equations, is performed when the stamp is defined. To model the component, means actually to determine the stamp. Of course, AleC++ hides many formal aspects of such model description from the user, so the user has to describe the model equations, rather than the stamp itself.

One can conclude easily that the stamps of linear time-independent components are constant throughout the simulation. Therefore, these calculations should be performed out of both simulation loops. Time-dependent linear components would have stamps that change in every time instant, so these calculations should be pulled out of the iteative loop. Finaly, nonlinear components have stamps that has to be recalculated in every iteration. For that reason, model designer can use `process` synchronization, as described in Chapter 5.

There are no many language constructs that are specific for analogue simulation. Chapter 5, giving general overview, applies in full. Analogue circuits can be described structurally, combined with a functional description, or using purely functional description (command `eqn`).

7.1. Built-in analogue models

Alecsis has a set of built-in SPICE-like models. Alecsis should not be used as a replacement of SPICE, but rather for simulation using new models, that are not available in SPICE. However, it was useful to have a set of built-in models, that can be used for SPICE-like simulation, and also as a basis for structural and combined structural-functional modelling.

7.1.1. Resistor, capacitor, inductor, and ideal sources

Basic components are **resistor**, **capacitor**, **inductor**, **ideal current** and **ideal voltage source**. All such components have two terminal nodes (in AleC++ terminology, two terminal links of type `node`). In Alecsis, they have only one parameter, named `value`, representing resistance, capacitance, inductance, current or voltage, respectively. That means, they are defined without the model card.

These elements allow shorthand notation.

```
resistor r1, r2;
capacitor cload, c2;
cgen i1;
vgen vdd;
inductor l1;

r1 (n1, 0) value = 3.4k;        // 3400 ohms
r2 (n1, n2) 2.7k;               // short of value = 2.7k
c2 (n2, 0) { value = 2.7k; }    // long, but legal
i1 (0, n2) 1mA;   // current has direction from 0 to n2
vdd (0, n3) 5v;   // 0 is 'positive' terminal, voltage of n3 is -5V
l1  (n1, n5) 5uH;
```

In the description given above, you can see that such parameters allow shorthand notation, where the keyword `value` is omitted. Nodes `n1`, `n2`, etc., represent unknown quantities in the system of equations.

Note: Node name `0` (number zero) is reserved for ground node.

Note: For ideal voltage source modelling and the inductor modelling, the branch current, i.e. current through the element is necessary. Therefore, every time when we connect such an element, a new link of type `current` is created. Such current carries the name of the element (`vdd` and `l1` are two currents in the example above). Creation of such new link, that is also an unknown quantity in the system of equations, is hidden from the user. However, one might need that current somewhere else -- as a result in command `plot`, or as a controlling variable in some other model. It can be accessed as any other link, using its name. This is the same as in SPICE.

We wanted to have user-defined models that have same characteristics as built-in components. It is already mentioned in Chapter 5, that **the `module` can return link under its name**. This is obviously intended for modelling of different voltage generators, where the current flowing through the branch can be returned to the hierarchically higher level. That current carries the name of the `module`.

In the example above, declarations can be omitted if we use implicit declarations given in standard header file `alec.h`. Without the declarations, the above code is very similar in structure to SPICE input file.

From the components given above, resistor, ideal current and ideal voltage source are linear time-independent models. Capacitor and inductor are linear, but time-dependent models, as time derivatives appear in

model equations. Time derivatives are discretized using formulae given in Chapter 5, where the time-step size is changing during the simulation run.

7.1.2. Built-in signal generators

Alecsis does not have many built-in signal generators. This is not a drawback since the user can define new signal generators easily. Some models already exist in model library. For such description, one can use ideal voltage and current source as a structural basis, and can add signal function easily in the functional part of the model - an example of combined structural-functional modelling. Such models are used the same as built-in models.

For example, pulse generator is not an built-in component, although an unavoidable element of time simulation. It is described as a `module` whose declaration is stored in a standard library "alec.h", and the body in the appropriate library (alec.ao). You can find more details about this model in the appendix on standard libraries of AleC++.

Actually, built-in signal generators are here for historical reasons, as they were needed before language AleC++ was completed.

7.1.2.3. Piecewise linear signal generators

Piecewise linear (PWL) sources are built-in components in Alecsis. There are voltage and current piecewise linear generators, declared as **`vpwl`** and **`cwpl`**.

It is defined time-value pairs, as in this example:

```
vpwl vin;

vin (n1, 0) { 2, 3;  4, -4;  6s, -4V;  8.0s, 1; }
```

The waveform of this voltage generator is given in Fig. 7.1.

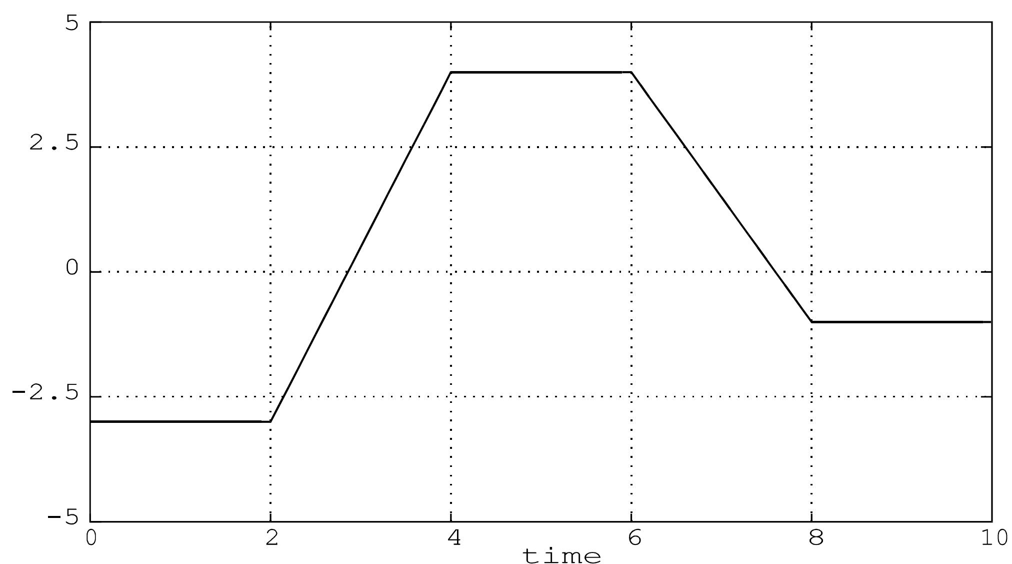


Figure 7.1. Waveform of piecewise-linear voltage generator, given by 4 time/voltage pairs.

The waveform is given by time/voltage pairs. The pairs have to be enclosed in parentheses, regardless of the number of points.

If the time instant of the first pair is greater than 0, the voltage value assumes the constant value between *t*=0 and the first pair, as in the example above. If the last time/value pair is before the `tstop`, the voltage keeps the constant value until `tstop`. Therefore, if there is only one pair, such voltage generator behaves as ideal voltage source of constant value. Similar rules apply for the current generator of type `cpwl`.

As any other voltage generator, `vin` in the example above generates link of type `current`. The current carries the same name, `vin`.

The number of parameters of such generator is not fixed. However, it need not to be a built-in element. AleC++ allows realization of a `module` where the `action` block has variable number of parameters (see Chapter 5 for details):

```
module new_vpwl(node i,j) action(double time0, double value0, ...);
```

You need to use alternative syntax of determining of actual parameters (similar to function calls) and to define some terminator (e.g., time moment `-1`), which will mark the end of the list. The call of such a component would be:

```
#define EOL -1.0
new_vpwl vin;

vin(n1, 0) action(0ns, 1v, 10ns, 2v, 100ns, 2v, 110ns, 2.5v, EOL);
```

Keep in mind that compiler cannot check the type of parameters in the place of symbol '`...`'. That is, all parameters, except the first two, have to meet expected types (`double` in this case). See Chapter 5 for details on implementation of such `action` blocks.

7.1.2.4. Sinusoidal signal generators

Alecsis has built-in voltage and current source of sinusoidal signal **`vsin`** and **`csin`**. The parameters are **`amp`** (amplitude in V or A), **`freq`** (frequency in Hz), **`phase`** (initial phase shift in rad), and **`dc_offset`** (DC component in V or A):

```
vsin vs;

vs  (n2, 0) { amp=2V; freq=1kHz; phase=30rad; dc_offset=0.5V; }
```

As any other voltage generator, `vs` in the example above generates link of type `current`. The current carries the same name, `vs`.

7.1.3. Controlled sources

Controlled sources are implemented as built-in elements, too. The controlling quantity at the input is either current or voltage, and the output quantity can be also either current or voltage. Therefore, there are four combinations:

- **`vcvs`** - voltage-controlled voltage source (takes parameter **`gain`**)
- **`ccvs`** - current-controlled voltage source (takes parameter **`mi`**)

- **`vccs`** - voltage-controlled current source (takes parameter **`gm`**)
- **`cccs`** - current-controlled current source (takes parameter **`beta`**)

```
vcvs vg1;
ccvs vg2;
vccs cg3;
cccs cg4;

vg1 (n1, 0, n3, n4) gain = 2;
vg2 (n5, 0, vg1) mi = 1.2;
cg3 (0, n2, n3, n4) gm = 1e-4;
cg4 (0, n2, vg2) beta = 50;
```

The first two actual links in all examples above are of type `node`. Voltage source `vg1` has `n1` as positive terminal, and `vg2` has `n5` as positive terminal. Currents of both current sources `cg3` and `cg4` assume direction from `0` to `n2` as positive.

Current-controlled sources `vg2` and `cg4` have one more actual link of type `current`, which is the controlling variable. Voltage-controlled sources have two actual links of type `node` for control. Both sources `vg2` and `cg3` are controlled by the difference of voltages at nodes `n3` and `n4`.

The example shows that voltage sources of type `vcvs` and `ccvs` generate new currents. They are here used for control of generators of type `ccvs` or `cccs`.

7.1.4. SPICE-compatible nonlinear components

Circuit simulator SPICE has become a standard for circuit simulation. The biggest quality of this simulator are the models of nonlinear electronic components. For that reason, we have implemented the same models (as built-in models) in Alecsis.

For these models, Alecsis accept unmodified SPICE model cards. The keyword `spice` is used for that. For instance, if file `mosfet.mod`, located in the working directory, contains models of MOS transistor given in SPICE syntax, it can be included in your description using:

```
spice {
# include "mosfet.mod"
}
```

7.1.4.1. Diode

Declaration of semiconductor diode utilizes the key word **`diode`**. A diode has two nodes -- anode and cathode, given in that order, and a model card of class **`d`**. The list of model card parameters, together with their default values, is given in Appendix.

```
diode d1, d2;

d1 (n1, 0) model = _1n914;
d2 (n2, n1) model = by238;
```

7.1.4.2. MOS transistor

Declaration of **m**etal-**o**xide-**s**emiconductor **f**ield **e**ffect **t**ransistor (MOSFET) utilizes the key word **`mosfet`**. MOS transistor has four terminal nodes -- drain, gate, source, and bulk, given in hat order, which is adopted from SPICE. It also accepts a model card of class **`nmos`**/**`pmos`** (the same model parameters, but for *n* and *p* type of transistor channel).

Alecsis supports 4 types of MOS transistor models. There are three SPICE2 models (model parameter `LEVEL` is 1-3), and also a BSIM model (**B**erkeley **s**hort-channel **I**GFET **m**odel) for transistors with submicron dimensions. According to HSPICE classification, BSIM model cards have the `LEVEL` parameter set to 13.

Regardless of the level, MOS transistors have geometric parameters for length (**`l`**), width of the channel (**`w`**), area (**`ad`**, **`as`**) and circumference of drain and source (**`pd`**, **`ps`**). Channel length and width must be given, while other parameter have default value of 0.

```
mosfet m1, m2;

m1(n1,n2,0,0) { model=nes2mos; l=2u; w=6u; ad=as=10p; pd=ps=40u; }
m2(n3,n4,n2,0) { model=pes2mos; l=w=3u; }//other parameters are 0
```

Note: MOS transistor models `LEVEL` 1, 2, and 3 have parasitic capacitances implemented as nonlinear capacitors, whose capacitances are calculated for given terminal voltages. This is so-called Meyer model, which exhibit charge nonconservation. Only BSIM (`LEVEL=13`) model has correct parasitic capacitance model, where charge conservation is guaranteed. In this model, terminal charges are calculated rather than capacitances, and parasitic currents are derivatives of these charges. Terminal charges can (optionally) appear as independent variables in the circuit system of equations (see section on `options` in Chapter 5).

Any SPICE manual will offer additional information on the names of parameters in cards, their typical values, and the appropriate equations for different levels. We give the list of model parameters and their default values in the Appendix.

7.1.4.3. Bipolar junction transistors

For declaration of **b**ipolar **j**unction **t**ransistor, keyword **`bjt`** is used. There are four terminal nodes -- collector, base, emitter, and substrate, given in that order. The last node, substrate, can be omitted. It accepts standard SPICE model card of class **`npn`**/**`pnp`**. Beside the model card, there is an optional parameter, **`area`** (area of the transistor).

```
bjt q1;

q1 (c, b, e) model = bc1107a;
```

The list of model card parameters, together with their default values, is given in Appendix.

7.1.4.4. JFET

For declaration of **j**unction **f**ield **e**ffect **t**ransistor (**JFET**), keyword **`jfet`** is used. It has three terminal nodes -- drain, gate and source, given in that order. It accepts standard SPICE model card of **`njf`**/**`pjf`** type. Parameter **`area`** (area of the transistor) is optional.

```
njf j1;

j1 (d, g, s) model = J2N2068;
```

The list of model card parameters, together with their default values, is given in Appendix.

7.1.5. Ideal switch

Switch is implemented in Alecsis as ideal component, whose resistance is zero when the switch is closed (*on* state) and infinite when it is open (*off* state). In Alecsis, switch is a voltage-controlled component. Unlike other ideal switches this does not pose any limits in circuit topology. Even loops and cutsets of switches are allowed, providing that the switching is regular. It can be used in both linear and nonlinear circuits. Switch is itself modelled as nonlinear.

Note: SPICE switch model has finite *Ron* and *Roff* resistances. Therefore, our switch model is not compatible with SPICE. You can easily define your own SPICE-like model of non-ideal switch, as a resistor whose resistance value is changed inside `action` block (see Section on combined structural-functional modelling on how to do that).

Switch is declared using keyword `switch`.

```
switch sw;

sw (n1, 0, n2, 0) { hyst=1; val_on = 3.5v; val_off = 1.5v; }
```

Switch has four terminal nodes. The first two are contact nodes, connected by the switch (`n1` and `0` in the example above). The last two are controlling nodes - voltage between two nodes is the controlling voltage *Vc*. In the example above, switch is controlled by the voltage at node `n2`, as 0 represents the ground node.

The switch has four parameters - **`val_on`**, **`val_off`**, **`hyst`** and **`paststate`**. The first two are the thresholds, which are compared to the controlling voltage *Vc*. In most of the applications, these two thresholds are equal, `val_on` = `val_off`. If *Vc*>`val_on`, the switch is closed. If *Vc*<`val_off`, the switch is open.

If `val_on` = `val_off`, parameter **`hyst`** plays no role. In the example above, you can see that `val_on` can be different than `val_off`. When *Vc* is between these thresholds, the behaviour of the switch is determined by the parameter `hyst`. This parametercan take values `0` and `1`. If `hyst` is `1`, the switch has hysteresis. The switch state if *Vc* is between `val_on` and `val_off` depends on the switch history. It is allowed both that `val_on` > `val_off`, and `val_on` < `val_off` in that case. If `hyst`=1, and `val_on` > `val_off`, the controlling of the switch is the following:

- when *Vc* is growing, when it passes the threshold `val_on`, the switch is turned *on* (closed);
- when *Vc* is decreased, when it passes the threshold `val_off`, the switch is turned *off* (closed).

If *hyst*=1, and `val_on` > `val_off`, the control is somewhat different:

- ♦ when *Vc* is growing, when it passes the threshold (whatever comes first, `val_on` or `val_off`) the switch is turned *on* (closed);
- ♦ when *Vc* is decreased, when it passes the threshold (whatever comes first, `val_off` or `val_on`), the switch is turned *off* (closed).

If parameter `hyst` is `0` (which is the default value), the switch has no hysteresis. In this case, `val_on` must be equal or greater than `val_off`. Between `val_on` and `val_off`, the switch has continuous change of resistance. **It should be noted that this is a continuous change between 0 (for `val_on`) and ∝ (for `val_off`).**

Note: Parameter `hyst` is optional, its default value is 0 (no hystersis).

Swtich also has parameter **`paststate`**. This is also an optional parameter. This parameter is actually not used to pass information to the model. It is intended to be used in another direction -- it returns information about the switch state in the previous (last solved) iteration.

In Alecsis, the switch is implemented as internally controlled, i.e. it is controlled by some circuit voltage. As that voltage can change from iteration to iteration, it is clear that the final state of the switch can be determined only when convergence occurs. In some cases, we want to know that switch state in the new iteration, and the parameter `paststate` can be used for that. For instance, if we model the diode *D* as the ideal switch, the model can be described as:

$$D:\begin{cases} closed, & if\ p>0 \\ open, & if\ p<0 \end{cases}, \qquad p=\begin{cases} i & if\ D\, is\, closed \\ v & if\ D\, is\, open \end{cases}, \tag{7.1}$$

where *i* is the current through the diode (switch), and *v* is voltage on the diode (switch). The diode can be controlled by voltage or current, depending on the diode state. To model it correctly using ideal switch, we have to know the previous state of the switch.

```
module switch_diode (node a; node k) {
   vgen vcaux;
   switch sd;

   vcaux (aux,0);
   sd (a,k,aux,0) { val_on=val_off=0.5; }

   action () {
      process per_iteration {
         if ( !sd->paststate ) {   /* switch was on (closed) */
            if((current) sd>0)  vcaux->value=1;
            else                vcaux->value=0;
         } else {                  /* switch was off (open) */
            if((node)a<(node)k) vcaux->value = 0;
            else                vcaux->value = 1;
         }
      }
   }
}
```

This is an example of combined structural-functional modelling that will be explained later in more details. The structural part of the model introduces the switch `sd`, and the ideal voltage source `vcaux`, which is used for switch control. (For diode model, we need switch controlled by the current. Switch is, however, implemented as voltage-controlled element, so we need some auxiliary voltage (node `aux`) that will reflect changes of current). In the structural part, generator `vcaux` is left without voltage value. This value is given in the functional part (`action` block), as an implementation of formula (7.1). Parameter `paststate` is used to return state of the switch in the previous iteration.

***Note*:** The current through the switch is introduced as a new quantity in the system of equations (as was the case with ideal voltage generators). In the above example, current `sd` was used in the action block.

We have tried to diminish differences between built-in component models and user-defined models (modules). We have explained in Chapter 5, that the `action` parameters can be used bidirectionaly. As you can see from this example, this is also the case with parameters of built-in components. Indirection operator `->` was used to access both the action parameters and the built-in component parameters.

The switch model was implemented as built-in, as it has some influence to time-step control. In some classes of circuits, it was very important to simulate the switch *just before* the switch transition, and *just after* the switch

transition. It is important to simulate it *just before* the transition to obtain exact capacitor charges and inductor fluxes in the moment of transition, as these quantities are of importance for the circuit after the transition. After the transition, the time-step is reset, since the circuit has new topology. As the switches in Alecsis are internally controlled, the exact switching instant is not known in advance. An iterative process is implemented to find the switching instant with desired accuracy. See section on `options` in Chapter 5, where options `SC_vtol`, `SL_itol` and `SDDT_tol` are described.

Note: We have already stated that ideal switch model can be used in any circuit topology, even when there are loops of switches and ideal voltage generators exist, or cutsets of switches and ideal current generators. The condition is that the switching is regular. If it is not the case, the solver cannot converge, which should be information for the user that there is problem with switching consistency (e.g. the part of the circuit is floating). However, you should be aware that options for difficult convergence, `dump` and `dcon`, can sometimes force convergence even in such cases. Therefore, they should be used with care.

7.2. Structural modelling

Many models can be described as connections of some other models, e.g. resistors, controlled sources, etc. If we have a `module`, that has only declarative and structural part, and no functional part (no `action` block), we consider that as a structural modelling. Both built-in components and user-defined modules can appear as parts of such model.

```
module real_voltage_source (node 1, 2) {
   resistor r1;
   vgen vg;

   vg(1,3) 5;
   r1(3,2) 1k;
}
```

Node 3 is implicitly declared in this example.

Purely structural model is actually describing subcircuit, defining circuit hierarchy in that way.

7.3. Combined structural-functional modelling

In fully functional modelling, the user can write model equations freely in the `action` block. However, this lack of restrictions can cause errors in the modelling process. Combined structural-functional modelling should be the preferred way of modelling, as it is more restrictive, and therefore not so error-prone.

In the combined approach, the user gives the structural description of the model (like in the structural modelling) but omit some or all of the parameters. These parameters are calculated and assigned to the components later, in the functional part (`action` block). Such modelling technique is often used in electronics. An example is modelling of a nonlinear component, for instance diode, by linearizing it. Linearized diode model consists of a resistor and ideal current source (Figure 7.1.). Parameters of these components R_d^m and i_{ds}^m are recalculated in every iteration m using expressions (7.2) and (7.3)

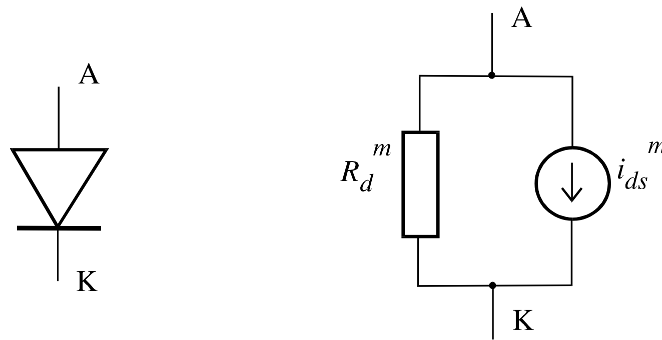


Figure 7.2. Diode and its linearized model.

$$R_d^m = \frac{1}{\left.\frac{\partial i_d}{\partial v_d}\right|_{v_d = v_d^m}} \qquad (7.2)$$

$$i_{ds}^m = i_d^m - \frac{v_d^m}{R_d^m} \qquad (7.3)$$

Therefore, combined structural functional model of diode would consists of a resistor and ideal voltage source connected in parallel, and an `action` block, where parameters of these components are calculated in every iteration.

There is another reason to use combined approach instead of purely functional modelling whenever possible: Alecsis can check circuit irregularities, such as loops of ideal voltage sources and inductors, or cutsets of ideal current sources and capacitor, and inform the user about them. Such checking is possible when we use combined modelling, since notion of these electronic components is still there. However, if we use equations, Alecsis cannot recognize components in them.

In combined modelling, you need to reference elements inside a process to access them. The name of an element in expressions is a pointer to the structure of element parameters. Some elements (resistor, capacitor, etc.) have only one element in the structure -- `value`, while the others have more (named) parameters. For instance, parameter `value` of resistor `r1` is accessed as `r1->value`. All built-in element parameters are of `double` type, and the parameters of user defined components (submodules) are given in their declaration.

The problem can arise with component that return current (or some other link), like voltage sources and inductors are, since the names of element represent the current at the same time. Confusion is avoided as we can get the current (or other link) using **cast** operation (for source `v1` the current is `(current)v1`). Without cast, the name represents the pointer to parameters.

7.3.1. Time-dependent linear models

We have already said that Alecsis has only ideal voltage and current source, and a pair of ideal signal generators as built-in components. Using combined modelling approach with ideal source as a basis, any signal generator can be easily described. To describe their behaviour, you often need only one process, and one line. For modelling of linear generators `process` with `per_moment` synchronization is used. The `process` is executed once in the given time instant.

```
#define twopi 6.282
module singen (node i, j) {
```

```
   vgen gen;
   gen (i, j);
   action per_moment (double amp=1.0v, double freq=1kHz,
                      double phase=0.0rad, double offset = 0v) {
      gen->value= offset + amp * sin( twopi*freq*now + phase );
   }
}

root test () {
   singen g1, g2;

   g1 (node1,node2) { amp=0.2V; freq=50Hz; phase=0; offset=0.5V; }
   g2 (node3,node4) action (0.2V, 50Hz, 0, 0.5V); // another way

   ...
}
```

This example shows the simplicity of modelling of sinusoidal generator with adjustable amplitude, offset, frequency, and phase (all these parameters have default values, as well). As there is only one `process` , the keyword `process` is not used, since the compiler takes the body of `action` with `per_moment` synchronization

Name of voltage source `gen` is used as pointer to structure containing parameters. There is only one parameter in the structure. That parameter can be reached using operator of indirection (`->`), i.e. as `gen->value`. In such case, when there is only one parameter in the structure, one can use indirection by dereferencing (operator `*`), too:

```
*gen = offset + amp * sin ( twopi * freq * now + phase );
```

Built-in elements have their signals for synchronization. This is why linear and time-independent elements behave as if they have processes sensitive to signal `initial`, in other words the contributions to the system of equations are calculated only once. **If we change values of parameters of such an element in the `process per_moment`, like we do with the voltage source `gen` in the example above, we change their synchronization to `per_moment`.** In the same way, we can change synchronization of linear time-independent and linear time-dependent models to per_iteration, if we change values of their parameters in a process `per_iteration`. Nonlinear elements (transistors, diodes, etc.) cannot change their synchronization , as they already have the most frequent refreshing, `per_iteration`.

Note: You may have noticed operator **`now`** inside the function `sin`. It simply returns the current time moment of the simulation, which amounts to 0 for `structural`, `post_structural` and `initial` processes. You can call it from any C/C++ -like function, but if the function has not been called during the simulation the value of the operator will be 0.

Note: In the structural description, parameter values need not to be omitted. If parameter value is assigned in functional description (`action` block), the value given in structural part is overwritten. The value assigned in the structural part is valid before the first execution of appropriate functional description. If no value is assigned, 0 is assumed until the first execution of functional description. It is useful to assign nonzero values in the case when zero values lead to singular matrix, which would abort the program in the phase of matrix renumeration (only processes with synchronization `structural` are executed before the renumeration).

7.3.2. Nonlinear models

All built-in nonlinear components are modelled in Alecsis using Newton-Raphson method of linearization. We can use Newton-Raphson method to linearize nonlinear models before composing equations, rather than to

linearize nonlinear equations. You need to create a linearized scheme representing nonlinear component, where linear components in that scheme change values of parameters in every iteration. Iterations are repeated until convergence is reached.

We have already described such method in this section, using diode model as an example (Fig. 7.2.). The parameter values of linear components are obtained by differentiating nonlinear functions. Therefore, model of any nonlinear component can be described by declaring and connecting components of the linearized circuit, followed by calculations of differentials in every iteration. Differentiation has to be performed with respect to all controlling quantities, i.e. quantities that appear in the original nonlinear expressions. Often, all terminal quantities of that nonlinear component are the controlling quantities.

We shall give here example of a MOS transistor. This is the simplest version of MOS transistor model, SPICE level 1 (Shichman-Hodges) model, without parasitic capacitances. Model is given as dependence of drain current I_D on all terminal voltages. This nonlinear model has to be linearized with respect to all controlling quantities. If we consider source *S* as referent voltage, there are three controlling voltages - V_{GS}, V_{DS}, V_{BS}. Linearized model is given as a parallel connection of three voltage-controlled current sources, and one independent current source (Figure 7.3.). Their parameters are recalculated in every iteration *m* using expressions (7.4.-7.7.).

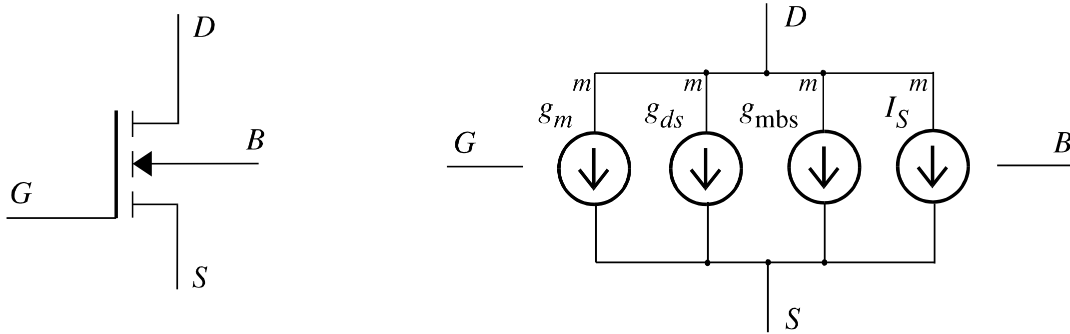


Figure 7.3. MOS transistor and its linearized model.

$$g_m^m = \left.\frac{\partial I_D}{\partial V_{GS}}\right|_{V_{GS}=V_{GS}^m, V_{DS}=V_{DS}^m, V_{BS}=V_{BS}^m} \tag{7.4}$$

$$g_{ds}^m = \left.\frac{\partial I_D}{\partial V_{DS}}\right|_{V_{GS}=V_{GS}^m, V_{DS}=V_{DS}^m, V_{BS}=V_{BS}^m} \tag{7.5}$$

$$g_{mbs}^m = \left.\frac{\partial I_D}{\partial V_{BS}}\right|_{V_{GS}=V_{GS}^m, V_{DS}=V_{DS}^m, V_{BS}=V_{BS}^m} \tag{7.6}$$

$$I_S^m = I_D^m - g_m^m V_{GS}^m - g_{ds}^m V_{DS}^m - g_{mbs}^m V_{BS}^m \tag{7.7}$$

MOS model has a number of technological parameters. Those parameters should be grouped in model card. A class containing these parameters is defined firstly. Model code is then described using functions that are methods of model class. These functions have access to parameters. Processes become shorter when bulk of the calculations is performed in methods.

The following example gives a functional description of MOS transistor:

```
#define Ntype 1.0
#define Ptype -1.0
```

```
class simple_mos {        // new class of model cards
        double type, gamma, phi;
        double uo, vto, lambda;
    public:
        simple_mos ();
        >simple_mos ();
        ~simple_mos ();
        friend module smos;
};

simple_mos::simple_mos () { // constructor sets the initial values
    type = 0.0 ;    gamma = 0.7 ;    phi    = 0.5 ;
    uo   = 0.06;    vto   = 1.0 ;    lambda = 0.0 ;
}

#define  max(x,y) ((x)>(y)?(x):(y))

module simple_mos::smos (drain, gate, source, bulk) {
    vccs e_gm, e_gmbs, e_gds;
    cgen e_iaux;

    /* linear voltage-controlled sources */
    e_gm   (drain, source, gate,  source) gm=0.0;
    e_gds  (drain, source, drain, source) gm=0.0;
    e_gmbs (drain, source, bulk,  source) gm=0.0;

    /* ideal current source */
    e_iaux (drain, source);

    action per_iteration ( double w, double l ) {
        double vto_mod, vd, vg, vs, vb, vds, vbs, vgs;
        double sarg,von,vgst,arg,beta,betap;
        double gm, gds, gmbs, ids;

        vd = drain;   vg = gate;   vs = source;   vb = bulk;

        vds = type * (vd - vs);          vbs = type * (vb - vs);
        vgs = type * (vg - vs);          vto_mod = type * vto;

        beta  =w/l*uo;
        if (vbs<=0.0)  sarg=sqrt(phi-vbs);
        else {
            sarg=sqrt(phi);   sarg -= vbs/(sarg+sarg);
            sarg=max(0.0,sarg);
        }
        von=vto_mod+gamma*(sarg -sqrt(phi));    vgst=vgs-von;
        if (sarg<=0.0) arg=0.0;
        else arg=gamma/(sarg+sarg);
        if (vgst<=0.0) {
            /*     cutoff region     */
            ids = gm = gds = gmbs = 0.0;
        }
        else {
            betap=beta*(1.0+lambda*vds);
            if (vgst<=vds) {
                /*    saturation region    */
                double vgst2;
```

```
                    vgst2= 0.5*vgst*vgst;     ids  = betap*vgst2;
                    gm   = betap*vgst;        gds  = lambda*vgst2;
                    gmbs = gm*arg;
                }
                else {
                    /*    linear region    */
                    double betap_vds,vdsh, arga;

                    betap_vds=betap*vds;
                    vdsh = 0.5*vds;
                    arga = vds *(vgst-vdsh);
                    ids  = (gm=betap_vds)*(vgst-vdsh);
                    gds  = betap*(vgst-vds)+lambda*beta*arga;
                    gmbs = gm*arg;
                }
            }
            /*    update element values    */
            e_gm->gm = gm;   e_gds->gm = gds;   e_gmbs->gm = gmbs;
            *e_iaux = type *(ids - gm*vgs - gds*vds - vbs*gmbs);
        }
    }

    model simple_mos :: simnmos { type=Ntype;
        vto = 0.82; gamma=0.59; phi=0.686; uo=0.051; lambda=0.051;
    }

    model simple_mos :: simpmos { type=Ptype;
         vto = -0.84; gamma=0.933; phi=0.733; uo=0.021; lambda=0.05; }

    root module test () {
        smos m1, m2;

        m1 (n_out, n_in, 0, 0) { W=5u; l=3u; model=simnmos; }
        m2 (n_out, n_in, n_Vdd, n_Vdd) { W=5u; l=3u; model=simpmos; }
        ...
    }
```

Note: In the structural description, parameter values need not to be omitted. If parameter value is assigned in functional description (`action` block), the value given in structural part is overwritten. This is the case with values for `gm` given in the structural part to sources `e_gm`, `e_gds` and `e_gmbs` in the example above. The value assigned in the structural part is valid before the first execution of appropriate functional description. If no value is assigned, 0 is assumed until the first execution of functional description. It is useful to assign nonzero values in the case when zero values lead to singular matrix, which would abort the program in the phase of matrix renumeration (only processes with synchronization `structural` are executed before the renumeration).

7.3.3. General nonlinear sources (automated linearization)

The linearization performed on MOS transistor model in the previous example means that we use Newton-Raphson method. The drawback is that the user is responsible to define both the structure of the linearized circuit, and to define partial derivatives with respect to all controlling variables.

To make description of nonlinear models less error-prone and more user-friendly, we have developed special nonlinear controlled generators, where the process of linearization is performed automatically. These are:

- `nlcgen` - general nonlinear current generator;
- `nlvgen` - general nonlinear current generator;
- `nlgen` - general nonlinear equation.

In these nonlinear controlled generators, the user supplies only the nonlinear dependence, and Alecsis estimates partial derivatives, replacing them with appropriate finite differences. Alecsis is able to calculate these finite differences automatically. That means that the secant method is used for solving nonlinear problems in that case. Newton-Raphson method has quadratic convergence (error in the next iteration is equal to the square root of the error in the previous iteration). Secant method has lower order of convergence -- 1.62, instead of 2. Therefore, order of convergence is decreased.

Note: If we have a nonlinear circuit, usually only some of the models are described using nonlinear generators `nlcgen`, `nlvgen`, and `nlgen`. Other, for instance, built-in models use Newton-Raphson linearization. Therefore, the actual decrease of the convergence rate is low, and depends on the problem.

Since Alecsis estimates partial derivatives numerically, the nonlinear dependence can be even non-differentiable (it is, however, understandable that non-differentiable functions with very strong nonlinearities would lead to convergence problems).

In all three types of nonlinear generators, there can be any number of controlling links (the only constraint is that there should be at lest one controlling link). Besides, type of controlling links is not constrained, too -- they can be of any analogue type (`node`, `current`, `charge`, and `flow`).

We shall now describe syntax of each type of nonlinear generators in more details.

7.3.3.1. Nonlinear current generator -- nlcgen

Nonlinear current generator has two nodes for connection, denoting where the generator current is flowing, and at least one controlling link (of any analogue type). The connection nodes are given first. The order of connection nodes is important, since it gives direction for current, as in any other current generator. The order of controlling links is arbitrary. If the connection nodes are also controlling (i.e. if the value of the current depends on connection nodes), they has to be stated again as controlling nodes).

In the structural part of the model, nonlinear generator is given without any parameters - only its connection is defined. The nonlinear dependance of the current is given in the functional description, i.e. in the `action` block. A `process` synchronized `per_iteration` has to be used, as dependence is nonlinear.

The example of MOS transistor model using **nlcgen**.

```
module simple_mos::smos (drain, gate, source, bulk) {
    nlcgen mos_current;

    mos_current (drain, source, drain, gate, source, bulk);
    /* current flows between drain and source, and is controlled
       by node voltages drain, gate, source and bulk */

    action per_iteration ( double w, double l ) {
        ...
        ... // calculates ids without derivatives - skipped here
        nlcgen mos_current = type * ids;
    }
}
```

The current flows between nodes `drain` and `source`, and the voltages of all for connections control it. As `drain` and `source` are controlling voltages, too, they are repeated in the set of controlling links. In the structural part of the `module`, no parameter is assigned to the nonlinear generator. The current value is calculated in the process synhronized `per_iteration`. In the same process, value is assigned to the generator using its name. In our example this was:

```
nlcgen mos_current = type*ids;
```

In simpler cases, the whole calculation can be performed on this assignment, so that the `process` contains only one line of the code.

Note: Assignment operand (=) must not be understood literally here. The current is not actually assigned a value, partial derivatives are calculated and the contributions are added to the system of equations (so-called model stamp).

There can be any number of nonlinear generators declared inside one `module`. However, in the action block of the module, each nonlinear generator value is calculated inside separate `process`. In every iteration, Alecsis executes that `process` code more than once, to estimate partial derivatives. Therefore, each nonlinear generator needs its own `process`, and all calculations regarding this generator should be in that `process`. **Therefore, there has to be one, and only one `process` per nonlinear generator:**

```
module two_generators (node n1, n2, n3; current c3; flow f4) {
    nlcgen gen1, gen2;

    gen1(n1, n2, c3, f4);
    gen2(n1, n3, n2, c3, n4);

    action () {
        process per_iteration {
           nlcgen gen1 = c3*f4;
        }
        process per_iteration {
           nlcgen gen2 = 5.*n2 - 6.*c3*n4;
        }
    }
}
```

Keyword `nlcgen` and component name (`gen1` or `gen2`) give information to Alecsis which `process` to connect with particular nonlinear generator.

Some special cases of controlled sources behave as basic components. So, `nlcgen` controlled by its own voltage behaves as a resistor:

```
module new_resistor (node n1, n2) {
    nlcgen r1;

    r1(n1, n2, n1, n2);

    action (double value) {
        process initial {
           if (value == 0.) warning ("zero resistance", 0);:
        }
        process per_iteration {
           nlcgen r1 = (n1-n2)/value;
        }

    }
}
```

You can also model easily current-controlled and voltage-controlled current source (`cccs` and `vccs`). However, in all these examples, synchronization `per_iteration` is used for otherwise linear and constant models, which makes such description inefficient.

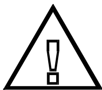
Nonlinear generator frees the user from determining structure of the linearized model and from calculating partial derivatives. Nevertheless, you can use nonlinear generators in a different way, with user-defined partial derivatives. In this way, only the structure of the linearized model is determined automatically. The user has to define partial derivatives, and the Newton-Rahpson method is used, with quadratic convergence. The following example demonstrates modelling of a diode:

```
module new_diode (node an, ch) {
    nlcgen Id;
    Id (an, ch, an, ch);
    /* current flows from an to ch, controlled by
       node voltages an and ch */

    action (double is=1e-14) {
        process per_iteration {
            double vt = 25.8mV;
            double gd, id;

            id = is*(exp((ah-ch)/vt) - 1);
            gd = (id + is)/vt;
            nlcgen Id = id { @an = gd; @ch = -gd; }
        }
    }
}
```

With this description, simulator knows that source `Id` has the value `id`, and partial derivatives with respect to controlling links `an` and `ch` in every iteration. Operator @ is used with controlling link name to denote partial derivatives of a nonlinear function with respect to a particular link. If you omit the block containing partial derivatives, the secant method is used:

```
...
nlcgen Id = id;
```

When partial derivatives are explicitely stated, the restriction about the number of nonlinear defined sources in a single `process` does not apply (the `process` is now executed only once in every iteration).



If at least one partial derivative is explicitly given, Alecsis does not estimate numerically other partial derivatives (which are not given). Therefore, they are undefined, which can cause simulation errors. Therefore, nonlinear generators have to be used with all partial derivatives (partial derivatives with respect to all controlling links) omitted, or with all derivatives explicitly calculated.

Alecsis does not give any warning about possible errors when only some of the partial derivatives are calculated. We will improve this in next versions.

7.3.3.2. Nonlinear voltage generator -- nlvgen

Nonlinear voltage generator has two nodes for connection, and at least one controlling link (of any analogue type). The order of connection nodes is important, since it gives orientation of the generator (positive node first). All other rules and restrictions are the same as for nonlinear current generator (`nlcgen`).

```
root module an_example_for_nlvgen () {
   nlvgen nonlinear_vgen;
   vgen v1, v2;
   resistor r;

   v1(n1, 0) 2;
   v2(n2, 0) 3;

   nonlinear_vgen(a,0, n1, n2);
   r(a,0) 1.k;

   timing { a_step = 1.; tstop = 100.; }
   // current through nlvgen is available under its name
   plot { node a;  current nonlinear_vgen; }

   action {
      process per_iteration {
         nlvgen nonlinear_vgen = n2*n3;
      }
   }
}
```

This is a very simple example with `nlvgen`, which here multiplies two voltages (2 and 3 volts), giving result of 6. **As for any other voltage source, current through `nlvgen` is introduced as new analogue link in the system of equations.** Here, current `nonlinear_vgen` is plotted out, and its value is 6mA.

Nonlinear voltage generator can be also used with user-defined partial derivatives, using the same syntax as for `nlcgen`.

7.3.3.3. Nonlinear equation -- nlgen

Nonlinear equation (`nlgen`) has somewhat different usage than `nlcgen` and `nlvgen`, although it follows the same syntax rules. It has no connection nodes, but has an arbitrary number of control links. It creates a new analogue link of type `flow`, which carries the same name as the generator. All contributions to the system matrix are in the same row, which corresponds to this new `flow`. For that reason, we can say that `nlgen` creates new (linearized) equation in the system matrix.

```
# include <math.h>   // includes declaration for sqrt function

module nlgen_test (flow f1, f2) {
   nlgen gen;

   gen(f1, f2); // f1 and f2 are controlling flows

   action per_iteration {
      nlgen gen = sqrt(n1*n2);   // new equation
   }
}
```

In this example, equation gen=$(n1*n2)^{1/2}$ is created, and is added (in linearized form) to the system of equations. Obviously, unlike nlvgen and nlcgen, generator nlgen does not correspond to any electrical element. As it creates new quantity of type flow (in our case, flow gen), it is to be used for modelling of nonelectrical problems.

Other rules and restrictions are the same as for nlcgen and nlvgen. Generator nlgen can be used with defined partial derivatives, too.

Note: As automatic linerization is very convenient for users, we would create other forms of automated linearization in the following versions of Alecsis. This wouls be automated linearization in purely functional modelling (command eqn).

7.3.4. Virtual synchronization of processes

It is already explained that combined structural-functional modelling is based on connecting some built-in, or previously modelled components, and than changing their parameters in the functional part of the description (action block).

All built-in components have their internal synchronization -- resistors are filled into the system matrix as processes initial, capacitors as processes synchronized per_moment, etc. When such built-in components are used in the structural part of model, their synchronization can be changed in the functional part. For instance, if linear and time-independent built-in component is used as part of the linearized model, their parameters are updated in every iteration when they are referenced in process per_iteration. Naturally, re-synchronization is possible only from less frequent to more frequent synchronization level (e.g. if nonlienar components are referenced in process initial, they would be still updated in every iteration).

On the other hand, user defined models have processes with fixed syncronization. For instance, we can have model described using fully functional modelling (explained later in this Chapter) with process initial:

```
module constant_component (node n1, node n2) {
   ...
   action (double value) {
      process initial {
         ...
      }
   }
}
```

When describing some other model using combined approach, we can use a component of type contant_component as submodel. However, the problem can arise, since synhronization is here fixed, and no re-synchronization is possible. We want user-defined models to behave in the same manner as built-in components, and to be used equally. To enable that, we can add word **virtual** before the word process:

```
module constant_component (node n1, node n2) {
   ...
   action (double value) {
      virtual process initial {
         ...
      }
   }
}
```

Such process will behave as ordinary process initial, if you set the component parameter value when connecting the component, and do not change it afterwards. However, if the parameter value is

changed in a higher-order `process` (`per_moment`, `per_iteration`), the `virtual process` will be re-synchronized to that more frequent synchronization level.

If parent component is digital (processes are sensitive to changes on signals), such child `virtual process initial` would be re-synchronized to `per_moment`. The parameter value would be then updated when the parent `process` activates (when there is change on signal the parent `process` is sensitive to). However, the child `process` would be executed in every time-instant (`per_moment` synchronization), in order to fill the analogue system matrix.

Processes sensitive to signals cannot have virtual synchronization.

7.4. Functional modelling -- eqn statement

In fully functional modelling, user can freely write the equations that contribute to the system of equations. Therefore, there are no restrictions in what is to be described as a model.

Note: Lack of restrictions makes functional approach very powerful, but also error-prone. Alecsis, for instance, check if there are any loops of ideal voltage generators and inductors, or cutsets of ideal current sources and capacitors, etc. However, in fully functional modelling, there is no information about the model structure, so such checking is not possible. Therefore, one can make an error in `eqn` statement and create singular system of equations.

Equations are written using command `eqn`. They are written in the processes of the `action` block. Therefore, structural part of the model can be completely omitted. There are three basic forms of this command:

- *simple* `eqn` statement;
- *through* `eqn` statement;
- *across* `eqn` statement.

Simple `eqn` statement defines a single equation. All contributions to the matrix are in the same row.

Through `eqn` statement defines the current flowing through the branch between two specified nodes. It has contributions in two rows, corresponding to these two nodes, and can be replaced by two simple `eqn` statements.

Across `eqn` statement defines the voltage across the branch, between two specified nodes. It has contributions in three rows, corresponding to these two nodes and to the current flowing between them. Therefore, an across `eqn` statement can be replaced by three simple `eqn` statements.

Electrical current is therefore a *through quantity*, while the voltage is an *across quantity*. Such approach can be used in other physical problems, since we can define through and across quantities in them. Few examples are given in Table 7.1.

Table 7.1. Across and through quantities in different physical domains.

generalized quantities	electrical	mechanical - translational	mechanical - rotational	hydraulic	etc.
across quantity	voltage *V*	velocity *v*	angular velocity ω	pressure *p*	...
through quantity	current *I*	force *F*	torque τ	flow *Q*	...
power	*P=V I*	*P=v F*	*P*=ω τ	*P*=p Q	...

For across quantity, an equation equivalent to Kirchhoff Voltage Law is satisfied, while through quentities must satisfy an equivalent of Kirchhoff Current Law. If the designer of electrical or nonelectrical models uses such paradigm of modelling, consistent system of equations will be built by Alecsis. Through and across εθν statements are more restrictive than simple `eqn` statement, but they lead to better models.

Note: Using `eqn` statements, linear differential equations can be described. Nonlinear equations cannot be described directly. Linearization of the model, according to Newton-Raphson method, has to be performed by the user. Such linearization and `eqn` statement have to be in `process` synchronized `per_iteration`. In the following versions of Alecsis, we plan to introduce *nonlinear* `eqn` statement, that uses the mechanisms developed for nonlinear generators (`nlgen`).

7.4.1. Simple eqn statement

An example of simple eqn statement is the following:

```
eqn i: g*{i}-(2*v-8.)*{j}-4*{k}+5.-g*j=67.;
```

In this equation, contributions to the system matrix are defined. They are all in the row specified directly after keyword `eqn`, i.e. in the `i`-th row. The column where the contribution appears is given by the index in parentheses '{', '}', which multiplies the contribution. Expression `g*{i}` means that there is contribution `g` in the `i`-th column. So we have contribution (`-2*v+8.`) in the column `j`, and `-4` in the column `k`. All indices representing row and column, must be declared as analogue links (`node`, `current`, `charge`, or `flow`), so that a row and a column in a matrix corresponds to each of them.

Contributions that do not multiply any index in parentheses, are contributions to the right-hand side of the system of equations. So, in our example we have contribution to the right hand side of the row `i`, which is `(67-5+g*j)`. Note that `j` is here given without parentheses, which means that this is not a position in the matrix, but the number of type `double`, which represent the current value (last solution) of the analogue link `j`.

Note: The contributions to the columns must be before the symbol '=', while the contributions to the right-hand side vector can appear both before and after the symbol '='.

If the above equation is the only one that contributes to the row `i`, then the equation appear as such in the system of equations. Nevertheless, other `eqn` statement in the same or some other module, or any other built-in or user-defined model, can contribute to the same row. All these contributions are added to the row `i`, following the concept of "stamps" common in electronic simulation.

We can form a stamp for any model using `eqn` statement. Here is an example of resistor:

```
module new_resistor (node i, j) {
```

```
    action (double value=0.0) {
        process structural {
            if (!value) warning("zero valued resistor", 1);
        }
        virtual process initial {
            double g = 1/value;

            eqn i:  g*{i}-g*{j}=0;
            eqn j: -g*{i}+g*{j}=0;
        }
    }
}
```

The first `eqn` statement contributes to the row `i`, and defines the current flowing through the resistor branch from node `i` to node `j`. The same applies to the second `eqn` statement, but this defines current flowing from `j` to `i`.

Expression `g*{i}-g*{j}` is the current that is flowing out of `node i`. If `eqn i` would be the only one that contributes to matrix row `i`, we would have that this current is 0, which is senseless. But other components connected to the `node i` contribute to that `eqn`, and the complete equation is stating that the sum of all currents flowing out of `node i` is zero (Kirchhoff Current Law). **For such case, when we are modelling currents or voltages, it is much more readable, and less error-prone, to use through `eqn` statement or across `eqn` statement, described in the following sections. Simple `eqn` statement should be used only when we are not using Kirchhoff Laws or its equivalents described in Table 7.1., and that should be avoided if possible.**

Equation

```
eqn i: g*{i}-g{j}=0;
```

can be also written as:

```
eqn i: g*{i,j}=0;
```

which makes equations shorter.

If `i` and `j` are links of type `node`, this can be also written as:

```
eqn i: g*{i,j}.v=0;
```

Extension '`.v`' denotes voltage. In this case, Alec++ would check the type of `i` and `j`, and would exit and give an error message if they are not of type `node`. It can be also written as:

```
eqn i: g*{i}.v-g{j}.v=0;
```

If `i` and `j` are declared as links of type `flow` (rather than `node`), you can use extension .a. So, the appropriate equation would be:

```
eqn i: g*{i,j}.a=0;
```

or:

```
eqn i: g*{i}.a-g{j}.a=0;
```

Extension '.a' denotes nonelectrical variable of *across* type. Alecsis checks if i and j are declared as links of type flow.

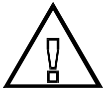
Note: Alecsis differs between electrical *across* quantity -- voltage (`node`) and electrical *through* quantity -- `current`; but with nonelectrical quantities, there is no such differentiation (in this version of Alecsis). All of them are of type `flow`, which can be used both as an *across* and as a *through* quantity.

As you can conclude from examples above, extensions '`.v`' and '`.a`' are optional, but are recommended, as they reinforce type checking. If extensions are not used, Alecsis checks only if both links in the pair `{m,n}` are links of the same type.

Note: Analogue links in `eqn` statement -- for instance, nodes i and j in statement:

```
eqn i: g*{i,j}.v=0;
```

must be scalars, as they are representing rows and columns in the system of equations. This applies for *through* and *across* `eqn` statement, too.



Analogue links in eqn statement can be scalars that are member of composite signals. Therefore, it is legal to define:

```
eqn w[2]: 5*w[2]-6*w[5]=32;
```

where `w[2]` and `w[5]` are scalar analogue links, members of link array `w[]`. However, it is not legal to write:

```
eqn w[m]: 5*w[m]-6*w[n]=32;
```

since link array indices must be constant.
This problem can be avoided with one additional hierarchical level in description. For instance, if you want to define:

```
process structural {
  for (m=1; m<=k; m++)
      for(n=1; n<=k; n++)
        eqn w[m]: 5*w[m]-6*w[n]=32;
}
```

you have to define an additional module for equation:

```
module Equation (node w1, w2) {
  action initial () {
     eqn w[m]: 5*w[m]-6*w[n]=32;
  }
}
```

and clone it in a loop, using `clone` command explained later in this Chapter:

```
module Equation Eqn;
...
process structural {
  for (m=1; m<=k; m++)
      for(n=1; n<=k; n++)
        clone Eqn(w[m],w[n]);
}
```

This applies on *through* and *across* `eqn` statement, too.

7.4.2. Numerical integration in eqn statement (ddt, d2dt2, idt)

Up to now, we have explained how to describe linear algebraic equations, that directly contribute to the system of equations. However, many physical problems need differential equations to be modelled. *Simple* `eqn`,

across eqn, and *through* eqn statements can be all modelled as differential equations. For example, capacitor modelled using equation:

$$i = C\frac{dv}{dt} \tag{7.8}$$

where i is the current through the capacitor, and v is the voltage accross its nodes, can be described as:

```
module new_capacitor (node i, j) {
    action (double value) {
        process per_moment {
            eqn i:  value * ddt{i} - value * ddt{j} = 0;
            eqn j: -value * ddt{i} + value * ddt{j} = 0;
        }
    }
}
```

Equation (7.1) is a typical through equation, so it is better to use *through* eqn statement. This will be explained in the next section.

Operator ddt stands for time derivative. It performs numerical integration (discretization). The numerical integration method is chosen in the options block, which is explained in Chapter 5. The way of filling the matrix depends on the method, but this is hidden from the user when operator ddt is used.

When operator ddt is used, contributions to the matrix is not constant - it depends on the time step, and on the system history (solutions in the previous time instants). For that reason, in the above example, eqn command is used in the process synchronized per_moment.

Shorter written is allowed here, too:

```
eqn i:  value * ddt{i,j} = 0;
```

as well as extension '.v' or '.a':

```
eqn i:  value * ddt{i,j}.v = 0;
```

The previous example of a capacitor can be modified, so that the current through the capacitor appears as unknown in the system of equations. The stamp is 'expanded' for one row and one column, and these carry the name of the current, which is here the same as the name of the component:

```
module current new_icap (node i, j) { // returns current on name
   action (double value) {
     process per_moment {
       eqn i:                           {new_icap} = 0;
       eqn j:                          -{new_icap} = 0;
       eqn new_icap: value*ddt{i,j}.v -{new_icap} = 0;
      }
   }
}
```

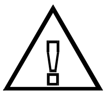
The first equation is stating that the current flowing outside of node i is new_icap, the second equation is stating that the current flowing outside of node j is -new_icap, and the third one is describing current/voltage dependence given by eqn. (7.8).

Capacitor model given by eqn. (7.8) can be rewritten as:

$$v = \frac{1}{C}\int idt \tag{7.9}$$

That can be described using **operator `idt`**, which **stands for integration with respect to time**:

```
module newer_icap (node i, j) {
   action (double value) {
      process per_moment {
         eqn i:                             {newer_icap}=0;
         eqn j:                            -{newer_icap}=0;
         eqn new_icap: {i,j}.v -1./value*idt{newer_icap}=0;
      }
   }
}
```



Operator `idt` is still not fully tested. Operator `ddt` is normally used for all SPICE-like modelling and simulation problems.

For application in modelling of electronic components, operator `ddt` (time derivative) is enough, as there are only first order differential equations. However, for modelling of mechanical systems, second time derivative is often necessary. One could avoid usage of second-order time derivative by introducing additional equation. Nevertheless, for the sake of model readability and to reduce size of the system of equations, we have introduced **second-order time derivative `d2dt2`**:

```
process per_moment {
   eqn x:  m*d2dt2{x} + ro*ddt{x} + c*{x} - {F} = 0;
}
```

In this example of mechanical equilibrium `m` denotes the mass, `ro` is the friction resistance, and `c` is the spring constant.

Operators `ddt`, `d2dt2`, and `idt` are connected to time-step control. Time-step control parameters are given in Section 5.6.3.1. in Chapter 5.

Operators `ddt`, `d2dt2`, and `idt` cannot appear in arithmetic expressions outside of command `eqn`.

7.4.3. Through eqn statement

If we want to describe behaviour of a quantity that has *through* character (Table 7.1.), the simplest way to is to use through `eqn` statement. For example, resistor model from section 7.4.1. can be described as:

```
module new_resistor (node i, j) {
    action (double value=0.0) {
        process structural {
            if (!value) warning("zero valued resistor", 1);
        }
        virtual process initial {
            double g = 1/value;
```

```
                eqn {i,j}.i = g*{i,j}.v;
            }
        }
    }
```

Extension '.i' on the left-hand side of the equation denotes the current. The equation above means that the current flowing between `node i` and `node j` equals `g` times the voltage between `node i` and `node j`. Expression `{i,j}.i` reinforces type checking, as `i` and `j` must be links of type `node` (current can flow only between nodes.

The through `eqn` statement above is fully equivalent to two simple `eqn` statements used for resistor model in section 7.4.1. This equation can be also written as:

```
eqn {i,j}.i = g*{i}.v-{j}.v;
```

and to:

```
eqn {i,j}.i = g*{i,j};
```

which means that extensions on the right-hand side of the equation can be omitted. (They cannot be omitted on the left-hand side of through equation.) The above equation can be also given with two through `eqn` statements.

```
eqn {i}.i = g*{i,j}.v;
eqn {j}.i = g*{j,i}.v;
```

where expression `{i}.i` on the left-hand side denotes the current flowing out of `node i`.

Many electrical models can be naturally described by through equation. For instance, statement:

```
eqn {i,j}.i = I;
```

describes current source of value `I`. It is equivalent to two simple `eqn` statement:

```
eqn i: 0 = I;
eqn j: 0 = -I;
```

It is clear that the through equation is much more readable, whenever the model represents the current through the component as some function of controlling quantities.

An equivalent expression for nonelectrical quantities would be:

```
eqn {m,n}.t = g*{m,n}.a;
```

where extension '.t' denotes through quantity. The above equations means: through quantity flowing from `flow m` to `flow n` equals `g` times the difference of across quantities `m` and `n`. Extension '.t' reinforces type checking, so `m` and `n` have to be links of type `flow` (of across nature -- however, Alecsis does not differentiate between across flows and through flows).

All variations of through equation, which are given above for currents and voltages, are valid for flows, too.

On the right-hand side, both nonelectrical and electrical variables can appear:

```
eqn {m,n}.t = g1*{m,n}.a + g2{i,j}.v;
```

This equations states that the through quantity flowing between `flow m` and `flow n` is the function of across quantity between `flow m` and `flow n`, and of voltage between `node i` and `node j`.

Note: Extension on the left-hand side of the through `eqn` statement ('`.i`' or '`.t`') cannot be omitted. Only nodes or flows can appear on the left-hand side (it does not make sense to define the current flowing between quantities of type `current`). On the right-hand side, extensions ('`.v`' and '`.a`') can be omitted, but are recommended to reinforce type checking.

Note: Extensions '`.i`' or '`.t`' can appear only on the left-hand side of through `eqn` statement. It does not make sense to put expression `{i,j}.i` on the right-hand side of the equation, since current flowing between nodes `i` and `j` is not available as the solution of the system of equations. If link `k` is of type current (for instance, current through the voltage source named `k`), it can appear on the right-hand side of the equation, but without any extension (extension '`.i`' would be confusing, since `{k}.i` denotes current flowing out of `node k`). For the reasons explained in this *Note*, extensions '`.i`' and '`.t`' cannot be used in *simple* `eqn` statement at all).

Through `eqn` statement can describe differential equation, since operators **`ddt`**, **`d2dt2`** and **`idt`** can be used, too. Here is an example of capacitor model:

```
module new_capacitor (node i, j) {
    action (double value) {
        process per_moment {
            eqn {i,j}.i = value * ddt{i,j}.v;
        }
    }
}
```

The model given above is fully equivalent to the first example from section 7.4.2. You can note that through eqn statement given in this model clearly describe model given by eqn. (7.8).

7.4.4. Across eqn statement

As we are using nodal approach for solving the system of equations, we describe most of the models as through equations (current as function of voltages). However some of the models cannot be described in this way. For instance, all kinds of ideal voltage generators (independent and controlled) have to be described as voltage dependence on other quantities. For that reason, we need modified nodal approach, where current through the branch is added as an additional unknown in the system of equation, and branch voltage equation is added as additional equation. This can be easily described using across `eqn` equation.

An ideal voltage source can be described in this way:

```
module current new_vgen (i, j) {
   action (double value) {
      virtual process initial {
         eqn new_vgen, {i,j}.v = value;
      }
   }
}
```

Expression `{i,j}.v = value` gives the dependence of voltage between `node i` and `node j` (in this case, this is a constant voltage). In through equation, current through the branch has also to be specified, since this is the row in the system matrix where this equation is added. The name of the current is given after keyword `eqn`, before the equation itself is given. In our example, the name of he current is `new_vgen`, and this is also the name of the `module`, declared as the `current` (this *returning the* `current` *on the* `module` *name* will be declared later in this Chapter). It is important that this variable, representing the current through the branch, is declared as `current` before the functional description (`action` block).

Across `eqn` statement

```
eqn new_vgen, {i,j}.v = value;
```

from the example above has the same contribution to the system of equations as three simple `eqn` statements:

```
eqn i:                          1*{new_vgen} = 0;
eqn j:                         -1*{new_vgen} = 0;
eqn new_vgen: 1*{i} -1*{j}                   = value;
```

These two descriptions define the same model stamp.

Across equation:

```
eqn k, {i}.v = r*{k};
```

has the same effect as:

```
eqn k, {i,0}.v = r*{k};
```

since `0` is always representing ground node.

Across equation can be described for nonelectrical quantities (flows), too. An example is:

```
eqn k, {m,n}.a = r*k;
```

The quantity that represents difference of `flow m` and `flow n` equals `r` times quantity `k`. Links `m` and `n` have to be of type `flow`, and of *across nature*, according to Table 7.1. (However, there is no difference between across and through flows on declaration in this version of Alecsis). Link `k` has to be of type `flow` (of through nature).

Note: Extension on the left-hand side of the across `eqn` statement ('`.v`' or '`.a`') cannot be omitted. Only nodes or flows can appear on the left-hand side. On the right-hand side, extensions ('`.v`' and '`.a`') can be omitted, but are recommended to reinforce type checking.

Note: This *Note* is the same as for through eqn statement: Extensions '`.i`' or '`.t`' cannot appear on the right-hand side of across `eqn` statement. If link `k` is of type current (for instance, current through the voltage source named `k`), it can appear on the right-hand side of the equation, but without any extension.

Across `eqn` statement can be differential, i.e. operators **`ddt`**, **`d2dt2`** and **`idt`** can be used. We have already defined model of a capacitor, with the current through the capacitor is added as unknown to the system of equations, in section 7.4.2. (`module newer_icap`). This can be easily described using across `eqn` statement:

```
module newer_icap (node i, j) {
   action (double value) {
      process per_moment {
         eqn newer_icap: {i,j}.v = 1./value*idt{newer_icap};
      }
   }
}
```

This across eqn statement clearly describes model given by eqn. (7.9).

For the capacitor model, we do not need current through the capacitor as new unknown in the system of equation, so it is more natural to use through equation. However, to model inductance, we need that current, since voltage is dependent on the current:

$$v = L\frac{di}{dt} \tag{7.10}$$

The inductor model using across `eqn` statement is:

```
module current new_l (i,j) {
   action (double value) {
      process per_moment {
         eqn new_l, {i,j}.v = value*ddt{new_l};
      }
   }
}
```

7.5. Appointed simulation in a time-instant -- breakpoint

Alecsis works with variable time-step. Time step is changed to meet demands on numerical integration error. For that reason, we cannot know in advance which time-instants would be chosen for simulation. However, in some particular cases, we want to force Alecsis to perform simulation in some particular time-instants. Function **`set_bpoint`** is used for that. It sets a *breakpoint* for a given time-instant (i.e. appoints simulation for that time-instant).

For instance:

```
set_bpoint(now+Period);
```

appoints simulation for time instant that happens `Period` after current time (keyword **`now`** returns current simulation time). Simulation proceeds with normal time-step control until it approaches the breakpoint, and then the time step would be shortened to meet the breakpoint. That means, it will not be allowed to jump over the breakpoint.

Function `set_bpoint()` can be used inside a `process` of `action` block.

An application example can be found in definition of `module pulse` (in file `alec.hi` in subdirectory `sys` that is normally installed with Alecsis.). This is definition of voltage generator of trapezoidal waveform. If time-step is large, corners of trapezoidal waveform can be missed. The simulation results are still correct, but the waveforms that are plotted out are not always nice -- corners of trapezoidal waveform can be cut out. Function `set_bpoint` is used here to force simulation in corners of trapezoidal waveform.

Note: As simulation time is variable of type double in floating-point arithmetic, the appointed breakpoints cannot be met exactly, due to some rounding errors.

7.6. Returning the link using name of a module

We have already mentioned that some built-in elements have to be modelled using branch voltage equations. For these elements, current through the branch has to appear as new unknown in the system of equations. That current can be plotted out, or used as controlling current for other models, using the name of the built-in component. This is the same as in SPICE.

Our concept was that user defined modules can be used equally as built-in component models. For that reason, AleC++ allows that the module name return the current (similarly as functions return variables - but here, it is not the value of the current that is returned, but the actual position in the matrix).

We have already given such examples in the previous section, for instance, ideal voltage source model:

```
module current new_vgen (i,j) { // new_vgen is name of the current
   action (double value) {
      virtual process initial {
         eqn new_vgen, {i,j}.v = value;
      }
   }
}
```

Here, current new_vgen is not declared separately, but when the module name is defined, as:

```
module current new_vgen
```

Current new_vgen can be used inside that module like any other current. However, when this module is used to declare components of type module new_vgen, each component of that type returns the current on its name:

```
module Y (node i,j,p,q) {
   module new_vgen X;         // THIS DECLARES X AS CURRENT
   ccvs Z;              // a current-controlled voltage source

   X(i,j) action( 5);
   Z(p,q,X) mi=1;       // current X is used for controlling
}
```

If we model the above voltage generator in the following way:

```
module newer_vgen (i,j) {
   current k;                 // separate declaration for current
   action (double value) {
      virtual process initial {
         eqn k, {i,j}.v = value;
      }
   }
}
```

then no association between name of the module and name of the current is established. Voltage source model is correct, but it does not return any current under its name, and cannot be used in our module Y.

This explains how to return current using the name of the module when the current is declared locally, inside the given module. But what to do when this current is passed by another module? For instance, when current returned by our module new_vgen has to be returned by our module Y, too. An association can be established in such case:

```
module current Y (node i,j,p,q){ // Y is also an current,
   module new_vgen Y;            // and it is returned by new_vgen
   ccvs Z;

   return Y(i,j) action( 5);     // keyword 'return' is necessary
   Z(p,q,Y) mi=1;
}
```

In this example, module Y also returns current under its name. It is the current returned by module new_vgen, and for that reason, component of type new_vgen also carries the name Y. When the component Y is connected, keyword return has to be used before the name Y, to point out the association of the names. (Without keyword return, Alecsis would consider this as name redeclaration, which is an error.)

This works in the same manner if we use built-in component (e.g. vgen), instead of user defined module new_vgen.

The compiler will not allow association of more than one component with the name of a module.

If you consider this too complex, you can return the current using list of formal parameters:

```
module Y (node i,j,p,q; current X){ // formal param. X is current
   module new_vgen X;          // and it is returned by new_vgen
   ccvs Z;

   X(i,j) action( 5);
   Z(p,q,Y) mi=1;
}
```

Here, `module Y` does not return any current on its name, but it returns `current` using list of formal parameters. As we have already explained, formal parameters of a `module` can be used for bi-directional communication, unlike parameters of C-like function. They are passed by reference, not by value, as they are representing position in the system of equations. **Keyword `return` is here not used for association.**

More than one component cannot associate with the same formal parameter, because that would be a name redefinition in the same structural visibility area (every component has its own current -- for example, every inductor has a current flowing through it.) **It is not legal to associate with a formal parameter that is not a scalar**.

Note: Links of type `charge` and `flow` can be returned using name of the `module`, too. All what is here explained for links of type `current`, is also valid for `charge` and `flow`. However, this feature is introduced for links of type `current`, to make Alecsis compatible with SPICE.

7.7. Variable number of action parameters

The number of parameters in the header of the `action` block of the `module` can be variable. This is described in the Chapter 5, as it is the same for digital and analogue modules.

7.8. Modules with variable structure -- clone and allocate

Some complex models cannot be easily described using standard Alecsis syntax. Sometimes we want that model card parameters or `action` parameters determine not only the behaviour, but also the structure of the model. There are two cases when this can be necessary:

- When some subcomponents appear in the module structure conditionally, depending on the model card parameters of `action` parameters. This can be, for instance, the case with resistances connected in series on component terminals, that are usually optional; or with whole optional subcircuits, e.g. some compensation subcircuits for opamps, etc.

- To define regular subcomponent arrays of variable size. Some electronic circuits exhibit clear regularity. Therefore, one can describe an array of variable size, where the size is an user-defined parameter.

Command **`clone`** is used to describe such models. It defines cloning of the previously described or built-in component. Cloning can be executed only in processes synchronized as `structural`, since execution of command `clone` defines the structure of the circuit. As this command is used inside a `process (structural)`, it can be executed conditionally, or in a loop, depending on the user-defined parameters.

Note: Models defined using command `clone` cannot be classified as either structural or combined (structural-functional), since here execution of `process structural` determines structure of the model.

The syntax of command clone follows all rules for component coupling:

clone_command:
clone *body_of_command_clone*

body_of_command_clone:
component ;
{ component_list }

component_list:
component ;
component_list component ;

When we use command `clone` to create arrays of variable size, it is clear that the number of links (e.g. terminal nodes) cannot be known in advance, too. So we need a mechanism to handle arrays of links of variable size. That *dynamical allocation of link arrays* is enabled using command **`allocate`**.

We can see application of both `clone` and `allocate` on the following example of ring oscillator:

```
module inverter(node vout, vin, vdd, vss) {
   mosfet mup, mdown;

   mup   (vout, vin, vdd, vdd) { model=Mpmos; l=3u; w=8u; }
   mdown (vout, vin, vss, vss) { model=Mnmos; l=w=3u; }
}

module ring_oscillator (vout, vdd, vss) {
   inverter inv;
   node joint[auto];       // array of nodes (size not defined)

   action (int size = 3) {
      process structural {
         allocate joint [size-1];   // dynamical allocation
         int i;
         if (size < 3 || size % 2 == 0)
            warning("wrong number of inverters in ring", 1);
         for (i=0; i<size; i++) {
            if (i==0)
               clone inv[0] (joint[0], vout, vdd, vss);
            else if (i==size-1)
               clone inv[i] (vout, joint[i-1], vdd, vss);
            else
               clone inv[i] (joint[i], joint[i-1], vdd, vss);
         }
      }
   }
}
```

In this example, an MOS inverter is defined structurally, and that `module inverter` is used as component to be cloned into an array. In `module ring_oscillator` we have component `inv` of type `inverter` declared. We also have declaration of an array of nodes with the name `joint`. Command:

```
node joint[auto];
```

is declaration of one-dimensional array of nodes. Length of that array is not yet defined, it will be done using command `allocate`.

We can note that `module ring_ocillator` has no structural part, i.e. declared component `inv` of type `inverter` is not connected in the structural part.

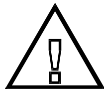
Inside the functional part (`action` block) a `process structural` is defined. In that process, array of nodes `joint` is allocated as:

```
allocate joint [size-1];
```

Array `joint` has length `size-1`, and `zero-offset` is used (from 0 to `size`-2). Parameter `size` is passed as an `action` parameter (default value is 3).

Note: Array of links can be have more dimensions, can have lower limit defined, etc. (all rules for Alecsis link array apply). Example:

```
allocate joint[1:size];        // array with offset 1;
```



When array of links has more than one dimension, only the first dimension can be dynamically allocated. For example, you can declare:

```
node a[auto][j];
```

but you cannot declare:

```
node a[auto][auto];
```

We intend to improve it in following versions of Alecsis, but as it is a complex intervention, it might wait for some time! (Array of links is very different than array of variables in C/C++ - links are positions in the system of equations.)
If you need to dynamically allocate array of links with more than one dimension, you can either reorganize it as one-dimensional array (which is always possible); or you can use array with static allocation, and recompile the model when you change dimensions:

```
# define FIRST_DIM 150
# define SECOND_DIM 200
...
node a[FIRST_DIM][SECOND_DIM];
```

In our example of ring oscillator, parameter `size` is checked firstly (it cannot be an even number, if we need an oscillator). After that, a `for` loop is defined, where `size` instances of component `inv` are cloned. A i-th instance of component inv is connected between nodes joint[i] and joint[i-1] using command:

```
clone int[i] (joint[i], joint[i-1], vdd, vss);
```

Index `[i]` after component name `int` can be omitted. However, it is useful, as it enables us to approach the particular component `int[i]` later -- in command `plot`, for instance.

This ring of inverters is a regular structure, but it has exceptions at the beginning, and at the end of the array (connections to output node `vout`). This is handled using `if/else` commands.

From this example one can conclude that both the component to be cloned, and the array of links of variable size, have to be declared in the declarative part of the `module`. The actual size of the link array, and cloning of components, are in the `action` block, `process structural`.

Note: Action parameters of cloned components can be different. However, if the component that is cloned has a model card, parameter `model` must be known in the compilation time (the model card name cannot be passed as a variable).

In command:

```
clone int[i] (joint[i], joint[i-1], vdd, vss);
```

index `[i]` after component name `int` can be omitted. However, it is useful, as it enables us to approach the particular component `int[i]` later -- in command `plot`, for instance. Index becomes a part of the component name, and in command `plot`, it must be used inside quotation marks, without any blanks inside the quotations:

```
plot { node s1/s2/"comp[5]"/n1;   current "lserial[3]"; }
```

In the above commands, we use cloned component name to define absolute path (to local link `node n1` in the component `"comp[5]"`); or to get the values of `current` returned under `module` name (`"lserial[3]"`).

Note: Commands clone and allocate apply to analogue, digital, and hybrid circuits in the same way, since nowhere in the syntax does the command demand a particular link type or component type. It is especially useful in digital simulation, for instance to describe registers of different lengths.

8. Hybrid simulation in Alecsis

If a circuit has both analogue and digital components, the simulation is hybrid. Analogue components are time-continuous, and analogue links (`node`, `flow`, etc.) have real values. That means, for analogue components system of equations is built and solved in many time-instants. Digital components are discrete-event, i.e. they are active only in discrete time instants. That means that an event-driven simulation algorithm is used, where propagation of events through the system is traced (no system of equations is necessary). Links in discrete-event components (`signal`) usually have discrete values (usually some enumeration type). However, in Alecsis signals can have real values (type double), too.

In hybrid simulation two kinds of coupling has to be performed:

- time-synchronization of analogue and discrete-event simulator;
- conversion of signals for all links with hybrid aspects.

Time-synchronization is performed by Alecsis, and user has no responsibility for it. However, second aspect of coupling demands user attention. Converters of signals are automatically inserted by the simulator for all links with hybrid aspect. However, Alecsis has not built-in system of states for digital (discrete-event) simulation. Therefore, user can define his own system of states. If he does that, he have to define his own D/A and A/D converters for that system of states, and these converters are later automatically inserted by Alecsis. For that reason, this Chapter concentrates on this second aspect of analogue/digital coupling.

Alecsis knows in advance only the nature of built-in components -- they are analogue. Other models, that are described in AleC++ or are already compiled in Alecsis object-code, can be analogue, digital or hybrid -- the syntax is the same. The nature of these models, and aspect of links (`analog`, `digital` or `hybrid`) can be determined only when the whole circuit description is read (the hierarchical tree, describing the circuit hierarchy is built).

The link has `hybrid` aspect if it is used both as analogue and as digital link. It is used as analogue if it appears as unknown in the system of equations. It is used as digital if some processes are sensitive to it, or it is driven by some digital driver. Between an analogue link and a component where appropriate formal signal has

direction `in` or `inout`, Alecsis inserts A/D converter. On the other hand, Alecsis inserts D/A converter between an analogue link and acomponent, if the component has a driver for the appropriate signal.

8.1. Implicit converters of link aspects

Alecsis automatically inserts a converter whenever it detects a hybrid link, but it cannot determine its structure. Converter structure depends on the system of states used for digital simulation, and of desired conversion accuracy. Therefore, converters have to be defined by the user. (Of course, this does not mean that you have to determine new converters for every new hybrid problem -- if you use some standard system of logic states from the library, and appropriate standard logic gates, you normally have standard A/D and D/A converters available in the library, too.)

Converters are just a special type of modules that user defines according to the manufacturing technology and the desired accuracy of the conversion. These modules must have two formal signals, one with the analogue, the other with the digital aspect. Converter cannot accept `action` parameters, but can accept parameters through the model card. However, converter has not its own model card, it accepts model card from the inserted digital component. To enable that, appropriate model class for converter has to be the same as the model class of the inserted component, or its base class.

Converters are modules of hybrid nature, since they have:

- formal links of both aspects;
- processes with digital synchronization (sensitive to signals) and analogue, synchronized by the internal simulator signals;
- analogue components declared in the structural region.

These analogue components are representing structural, or combined structrual-functional model. It is the model of the input of the digital component (in case of A/D conversion), or output of the digital component (in case of D/A conversion), as seen from the analogue part of the circuit. For instance, this can be only a capacitor marking input capacitance (for digital component in CMOS technology), or the input to a TTL circuit with a `bjt` and additional components. The number and the type of analogue elements depend on the technology and desired accuracy of conversion.

8.1.1. A/D conversion

A/D converters are inserted for every hybrid link that is connecting analogue components and the component with the appropriate formal signal having direction `in` or `inout`. A/D converter has two formal links - analogue link and digital signal. It is usually a combination of several elements modelling the input of a digital circuit; and a `process` where analogue link is compared with a series of fixed threshold, to determine state of appropriate digital signal.

This `process` is always synchronized using `post_moment` synchronization signal. It activates only when the analogue part of the circuit has an *accepted solution* for the present time-instant (analogue simulator reject solution in a time-instant if local truncation error is greater than the appropriate tolerance, and repeats the simulation with the shorter time-step). Using `post_moment` synchronization guarantees the validity of the analogue solution for the present time-instant.

It is already stated that the conversion inside that `process` comes down to comparison of an analogue quantity with a series of thresholds, which effectively converted a continuous quantity into discrete domain. The number of thresholds depends upon the system of states the converter is defined for. The values of thresholds can be

fixed, or depending upon parameters in case the converter has a model class. Usage of model class enables modelling of inputs in different technologies.

Conversion can be *direct* or *delayed*. In the first case, the values are converted as they come (e.g., if we have transition of a `node` from 0 to 5v, there is a passage through an zone in between thresholds `0` and `1` where converter will give output state `'x'`). The second method assigns the new signal only after it determines whether the transition to `'x'` state is a true undetermined state, or just transient phase `0->1` or `0<-1`.

Here is an example with *direct* conversion:

```
implicit { capacitor c, C; }

typedef enum { 'x', '0', '1' } three_t;

module cmos_a2d (node analog; three_t out digital) {
    capin (analog, 0) 0.1p;
    action post_moment {
        three_t last_state='x', new_state;
        if (analog > 3.5v) new_state = '1';
        else if (analog < 1.5v) new_state = '0';
        else new_state = 'x';
        if (last_state != new_state) {
            lasr_state = new_state;
            digital <- new_state;
        }
    }
}
```

This example presents a simple A/D converter intended for the connecting a hybrid link with the input of CMOS logic gate. The new event is generated in every time-instant of analogue simulation, but only after a threshold is reached. Assignment of signals in processes `post_moment` is processed in the same time-instant as analogue simulation (keyword now returns the same time for processes `per_moment` and processes `post_moment`, but the post_moment executes after the solution is accepted, i.e. local truncation error is small enough). No delay is generated. The digital circuit will not notice the difference between signals from the converters and other signals in the digital part (although you can check the predefined signal attribute `hybrid` to see whether the signal originates from digital component, oris generated by the converter).

The previous example uses direct conversion. The signals from the output of a digital circuit will have short intervals of `'x'` states in the transition period. This is a 'false alarm', although some logic simulators (e.g. HILO) have the capability for the logic circuits to generate those kinds of signals. Here is an example with *delayed* conversion:

```
module a2d (node analog; fift_t out digital) {
    capin (analog,0) 0.3p;
    action {
        process post_moment {
            fift_t last_event='x', new_event;
            double dval, last_val;

            dval = analog;
            if (dval >3.5) {
                /* rising !! */
                new_event = '1';
            }
            else if ( dval < 1.5 )  {
                /* falling */
                new_event = '0';
            }
```

```
                else {
                    /* is it real 'x' or just a transition? */
                    if (last_event == '0' && dval < last_val
                        || last_event == '1' && dval > last_val)
                        new_event = 'x';
                }
                if (last_event != new_event) {
                    last_event = new_event;
                    digital <- new_event;
                }
                last_val = dval;
            }
        }
    }
```

This converter uses not only the value of the analogue variable, but also the sign of the slope (the sign of the firs derivative) of the analogue curve. If the previous state was `'0'`, the state `'x'` will not be assigned immediately when the threshold for `'0'` is passed. The new state is not assigned while the slope is positive, i.e. during the transition. The same is valid if the previous state was `'1'`. State `'x'` is not assigned when the threshold for '1' is passed, while the slope is negative. When the first derivative changes the sign, state `'x'` is assigned. In this way, the normal transitions from one to another state do not generate `'x'` state, but only fluctuations, that are really undefined (`'x'`) states.

Note: Note that delayed conversion does not mean that any artificial delay is generated in the process of conversion. For instance, when the analogue signal is rising, state `'1'` is assigned immediately after threshold for `'1'` is passed. Name *delayed* means that the simulator waits to study the real behaviour of the signal during the transition only.

Converters for systems with more logic states can have more voltage thresholds. Also, converters for circuits with bipolar inputs need additional analogue components for correct modelling. However, the conversion procedure does not differ.

8.1.2. D/A conversion

For every digital driver of a hybrid link, D/A conversion is necessary. As was the case with A/D converters, compiler inserts D/A converters for appropriate digital component. That means, if we have an analogue component connected to the bus, driven by more digital components, mechanism of digital resolution will not be applied. D/A converters are generated for every driver, and the analogue simulator would resolve conflict on the bus.

The component with more than one driver for hybrid signal (more than one `process` can assign to given signal) would be then resolved in different way: digital resolution is performed inside the component, and the solution is converted into the analogue domain. This is usually not what is wanted, so the simulator issues warnings in those situations. Generally, in one digital `module` you should avoid creation of more than one signal driver (more than one `process` that assigns to one signal).

D/A converter is actually a controlled source. The analogue part of the converter should create the illusion for the analogue subcircuit that the whole circuit consists of analogue components. Model of output of digital component usually comprises a controlled source, output resistance, and output capacitance. These components can be linear or nonlinear, depending on the desired conversion accuracy.

The transition of state on the digital driver is abrupt. However, it should not be modelled as abrupt change of controlled source in a D/A converter. An abrupt change of the parameters of analogue component can create convergence problems, and, what is even more important, is not an accurate model of real behaviour. Abrupt changes of states in digital circuits are modelling of real circuit behaviour on the higher level of abstraction. In the

analogue part, we should use more accurate modelling, on the lower level of abstraction, where all changes are continuous. For that reason, good D/A converter ought to have two processes - one that sensitive to a digital signal, and the other to model continuous transitions.

This transition of the analogue controlled source in a D/A converter contributes to a total delay of a digital signal. Therefore, if the signal has attribute `hybrid` with value `1`, the delay assigned in modelling of digital circuit should be shortened for the transition time of the analogue source (so the total delay is correct). Beside that, delay due to load (see Chapter 6 on digital simulation, section on user-defined attributes of signals) should not be used, since the load is here analogue and the delay will be determined by analogue simulator. Therefore, **the delay, defined in digital processes for drivers of `hybrid` links, must not take into account capacitive load on the output, and is reduced for the time needed by the controlled source of the D/A converter to reach the half of the transition.**

Here is an example of D/A converter:

```
enum status { Rising, Falling, Steady };
#define NewR(state) (state=='x' ? 100k : state=='z' ? 1e9 : 1k)

#define RiseTime 10ns
#define FallTime 10ns
#define NewLevel(state) (state=='1' ? 5v : state=='x' ? 2.5v : 0v)
module cmos_d2a (four_t in digital; node analog) {
   resistor Rout;
   capacitor Cout;
   cgen Iout;
   Rout (analog, 0);
   Cout (analog, 0);
   Iout (0, analog);

   action {
      four_t old_state='0', new_state='0';
      status stat = Steady;
      double start_value=0, end_value=0, start_time=0, end_time=0;

      process (digital) {
         old_state = new_state;  new_state = digital;
         if (new_state == 'z' && old_state == '0' ||
            new_state == '0' && old_state == 'z' )
            stat = Steady;
         else if (old_state == '1' ||
                  old_state=='x' && new_state != '0')
            stat = Falling;
         else    stat = Rising;
         if (state != Steady) {
            start_value = NewLevel(old_state);
            end_value = NewLevel(new_state);
            start_time = now;
            end_time=start_time+(stat==Falling?FallTime:RiseTime);
         }
      }

      process per_moment {
         if (stat == Steady) {
            *Rout = NewR(new_state);
            *Iout = end_value/(*Rout);
         }
         else {
            if (now < end_time) {
```

```
                        *Rout=new_value=='z'?1e9:100k;
                        slope=(end_value-start_value)/end_time-start_time);
                        *Iout = (start_value+now*slope)/(*Rout);
                    }
                    else {
                        *Rout = NewR(new_state);
                        *Iout = end_value / (*Rout);
                         stat = Level;
                    }
                }
            }
        }
    }
```

The concept of this D/A converter is the in the recognition of the three states, which are enumeration constants `Rising`, `Falling`, and `Steady`. The `process` sensitive to change of the formal signal `digital` controls the state of the converter, while the `process per_moment` controls the values of analogue components according to the converter state. Analogue components are capacitor `Cout`, resistor `Rout`, and current source `Iout`. In the `Steady` state, these analogue components have fixed values, according to the last logic state. In the other two states current sources `Iout` changes its current according to the predefined equation. Converter can be transferred into the `Rising` or `Falling` state only by the first `process`, that is sensitive to signal digital. On the other hand, converter moves into the `Steady` state when the current source `Iout` reaches the final value.

The analogue simulator can reject the solution and backtrack, in order to shorten time step. However, this model is valid even if this happens (it will not backtrack beyond `start_time`).

We use current source `Iout` in parallel with resistance `Rout`, rather than voltage source in series with resistance. The latter would demand two more equations (one for the internal node for the series connection with the output resistance, and another for the current through the voltage source).

The drawback of this D/A converter model is that the model parameters are fixed. Such parameters should be grouped into model cards, as they depend on production technology.

8.2. Converter declaration

The simulator does not know in advance which converter models are to be used for given problem. You need to define this explicitly, whenever you expect a hybrid simulation. **Converter models are declared for digital circuits.** There are three ways of declaring converters:

– for the whole circuit (if technology is unified), in the `root module`;

– at the individual `module` level;

– for particular formal signal.

8.2.1. Converter declaration for the whole circuit

Declaration of the converter for the entire circuit is in the `root module`, after the structural region.

```
root module X () {
    ...     // declarative part
    ...     // structural part
    conversion { a2d = cmos_a2d; d2a = cmos_d2a; }
    ...     // printout control, timing and options control
    ...     // functional part
}
```

You create block for declaration of modules for A/D (parameter `a2d`) and D/A (parameter `d2a`) conversion using the key word **`conversion`** and parentheses. The names of modules can be also given together with the names of libraries they are stored in (e.g., `lib1.cmos_a2d`), otherwise linker will search all libraries.

When forming a hierarchical tree, the information on converters will be recursively passed to all children modules. The conflict with model cards will not arise if:

- children modules do not accept model cards, or
- the model class of the converter is the same as the model class of children module, or one of its base classes.

Converter declaration for the `root module` can be masked by declaration for children modules, or with declaration for individual formal signals.

8.2.2. Converter declaration for module

You can mask the converter declaration from the `root module` with an identical declaration in the current `module`. The new declaration applies to all submodules of that `module` (i.e to its children), if not separately masked in some of them. If only one converter is declared for the `module` (`a2d` or `d2a`), another one is obtained from the parent `module`.

Converter declaration for `module` can be masked by declaration for children modules, or with declaration for individual formal signals.

8.2.3. Converter declaration for formal signals

The declaration of converters is allowed for individual formal signals. This declaration has the priority. The of names of conversion modules are given using following syntax:

```
module rsff(three_t in reset:cmos_a2d, set;
            three_t inout q, nq: (cmos_a2d, cmos_d2a)) {
    ...
}
```

Converters are declared after the declarator (and after an optional initializer), and apply to only one signal (here only `nq`).

You can declare only one converter for formal signals with `in` direction (this must be A/D converter) and only one converter for formal signals with `out` direction (D/A). Here, signal `reset` with direction in has A/D converter declared. For `inout` signals, you can list both converters (in parentheses, separated by a coma, as for signal `nq`).

These converters are created as the children of the current `module`, and will be able to inherit its model card (under the conditions that model class of converter is the same as model class of the `module`, or its base class). Formal signals that do not have this declaration will inherit converters from the current module, or its parents (signals `set` and `q`). **Local signals cannot have converters in their declaration.**

8.2.4. Organization of class hierarchy for digital model classes

Since converters do not take `action` parameters or model cards directly, the only way the make them dependent on parameters is using the model cards of the parents. To use the same converter for digital circuits with different functions and parameters, you should do the following:

- You need to create the base class, which is accepted by the converters. It will be a common class for a large group of digital circuits, which makes converters common, too. You can place here all parameters that have no connections with the conversion, but are also common. To make them accessible from derived classes, you need to declare these parameters as `protected` and not `private`.

- For each type of digital circuits, that demands new, different parameters, you need to develop a new model class by **deriving** it from the **base** class. To avoid problems iwth offset, it is important for all derived classes to have a single class as base class.

The address of the derived model card will be the address of the *base part*, and so converters will be able to find their parameters. Initialization of the *base part* of the card can be achieved by defining a separate constructor. As Chapter 4 on object-oriented programming emphasizes, during a construction of a card (being a static object) the constructors of the base class will be invoked first, and then the ones from derived classes (vice versa for destructors). However, **base preprocessors are not automatic, and should be called from the derived processor**. All parameters of the base class should have usable initial values, which makes their redefinition necessary only in the case we want to change them.

```
class common {
   protected:
      int tech;                                    // basic technology
      double Cin, Rin;                             // impedance for A2D
      double Cout, Rout;                           // impedance for D2A
      double zero_level, unit_level, x_level; // analog levels
      double z_impedance;                         // turn-off impedance
      double rising_time, falling_time;        // transition times
   public:
      common();
      >common();
      friend module common_a2d, common_d2a;
};

class gate : public common {
      double tplh, tphl;
      double skew;
   public:
      gate();
      >gate();
      double f_delay(four_t new_state);
};

// define base constructor
common::common() {
   tech = CMOS; Cin=0.1p; Rin=1e15; Cout=0.1p; Rout=1k;
```

```
      zero_level=0v; unit_level=5v; x_level=2.5v; z_impedance=1e9ohm;
      rising_time = falling_time = 10ns;
   }

   // define base preprocessor
   common::>common() { ... /*anything*/  }

   // define derived constructor
   gate :: gate () { tplh = tphl = 10ns; }
   // define derived preprocessor
   gate :: >gate () { common::>common(); .../* enything ELSE */ }

   // define one model card
   gate :: mod1 {
      tplh = 3.4ns; tphl = 5.6ns;
      common::rising_time = 2ns; falling_time=3ns; tech = CMOS;
   }
```

The previous example defines `class common`, which stores common, mainly conversion data for a series of digital circuits. By declaring converters `common_a2d` and `common_d2a` as friends of the `class common` you enable them to access the `private` parameters. Setting those parameters in the constructor enables correct conversion, even if they are not set in the model card.

You can use the derived class `gate` for a larger number of standard logic gates, which use parameters `tplh`, `tphl`, and `skew` to model delay. Base class constructor is called before the derived class constructor automatically, while you have to call the base preprocessor explicitly form the derived class preprocessor.

Model card `mod1` of class `gate` can set parameters of both the base and the derived classes. You can access base parameters with or without the operator of access resolution (`::`).

This way of modelling allows modelling of new groups of digital circuits 'in layers'. Notice that every change in the base class automatically means changes in derived classes, according to the rules of object-oriented programming.

8.3. An example of hybrid circuit simulation

We left an example of simulation of parallel AD converter for the end of this Manual. Note that we are not speaking here about `module` for conversion of link aspects, but about the circuit for conversion of analogue signal into digital one. The ADC consisto of comparators, resistance network and combinational logic (Figure 8.1.).

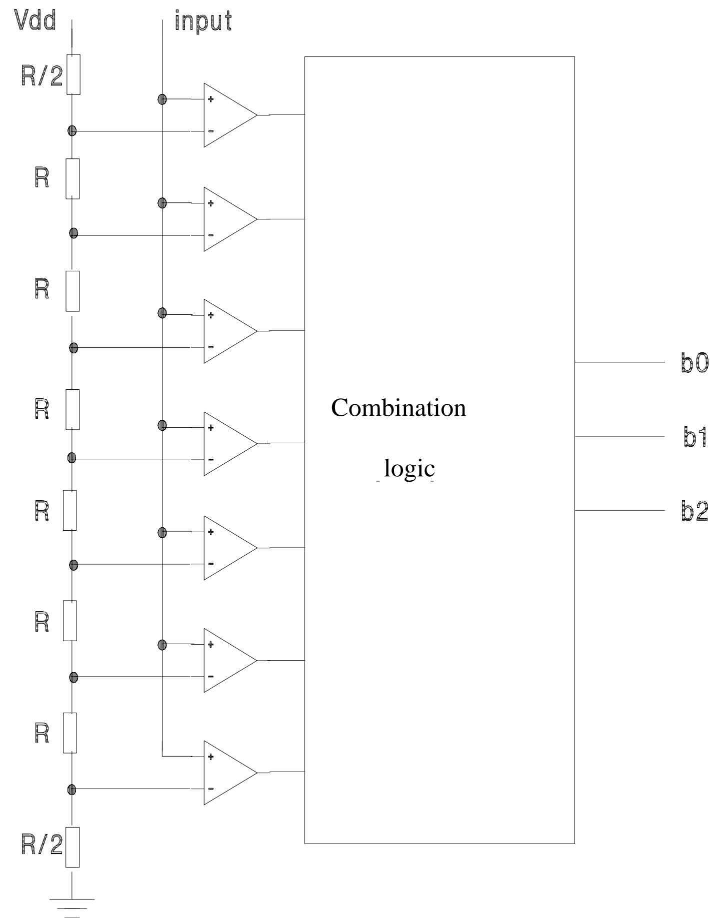


Figure 8.1. Parallel A/D converter as an example of hybrid circuit simulation

The circuit consists of seven CMOS opamps, which are used as comparators. Comparison etalons come from the serial resistance network. With the increase of input voltage, the outputs of comparators, one by one, change from the lower voltage supply level `vss` to the upper voltage supply level `vdd`, that is from 0 to 5V. The outputs of the comparators are directly fed to combinational network. The output of this digital network is a three-digit number. Input voltage of 0V corresponds to output `000`, and 5V to `111`. Every opamp has 15 MOS transistors, and is in cascade OTA configuration. Combinational network is modelled functionally, using overloaded logic operators. We have used libraries `opamp` (with amplifiers), `op15` (program support for system with 15 logic states) and `alec` (standard Alecsis library).

The nodes where analogue circuit and digital circuit meet are opamp outputs. These nodes are `hybrid`, and implicit insertion of converter of link aspects is performed there.

The code in full follows:

```
#include <alec.h>
spice {
#include "omos.mod"
}
library opamp, op15, alec;
#include "ss15.h"
#include "opamp.h"
/**************************************************************/
module comp_line ( node input; node line[], udd, uss) {
   module Ota opamp;
   resistor r, rup, rdown;
   node joint[auto];
```

```
    action structural (int resolution) {
       int i;
       node joint(resolution);

       if (resolution < 2)
          warning("illegal A2D converter resolution", 1);

       /* clone edge resistors */
       clone rup   (udd, joint [resolution-1]) Rvalue/2;
       clone rdown (uss, joint [0]) Rvalue/2;

       for (i=0; i<resolution; i++) {
          clone opamp [i] (input, joint [i], line [i], udd, uss);
          if (i<resolution-1)
             clone r[i] (joint[i], joint[i+1]) Rvalue;
       }
    }
}

/***************************************************************/
module comb_logic4 (fift_t in line[7]; fift_t out b[3]) {
    action (double delay = 10ns) {
       process (line) {
          three_t l0, l1, l2, l3, l4, l5, l6;
          three_t l23, all, b0, b1;

          l0 = Con15to3[line[0]];
          l1 = Con15to3[line[1]];
          l2 = Con15to3[line[2]];
          l3 = Con15to3[line[3]];
          l4 = Con15to3[line[4]];
          l5 = Con15to3[line[5]];
          l6 = Con15to3[line[6]];
          all = l0 & l1 & l2 & l3 & l4 & l5 & l6;
          l23 = l2 ^ l3;
          b1 = (l5 ^ l6) | l23 | (l1 ^ l2) | all;
          b0 = (l4 ^ l5) | l23 | (l0 ^ l1) | all;
          b[0] <- tech_tab[CMOS][Con3to15[b0]] after delay;
          b[1] <- tech_tab[CMOS][Con3to15[b1]] after delay;
          b[2] <- tech_tab[CMOS][Con3to15[l3]] after delay;
       }
    }
}

/****************************************************************/
#define Period 100us
#define Resolution 7
#define VDD 5v
root parallel () {
    vpwl vin;
    comp_line comparator;
    comb_logic4 combinatorial_logic;
    vgen vdd;
    fift_t line[Resolution];
    fift_t b[3];

    comparator (input, line, udd, 0) resolution = Resolution;
```

```
    combinatorial_logic (line, b);
    vin (input, 0) { 0,0v; Period, VDD; };
    vdd (udd, 0) VDD;

    conversion { a2d = "a2d"; }
    plot      { node input;  node line;  signal fift_t b; }
    timing  { tstop = Period; a_step = Period/100; }
    options { dcon=1; maxiter = 20; maxdump = 5; }
}
```

The results of hybrid simulation of this circuit are given in Figure 8.2. We have obtained correct conversion. You can notice the shift in the average signal value at the input of the opamp (increase in voltage on the '-' input), which changes its characteristics (reduces the slope), and that introduces distortion into the converter characteristics. The whole circuit contains 105 MOS transistors. To solve it, Alecsis forms a system of 103 equations, seven `post_moment` `process`es, and one digital `process` (for combinational logic).

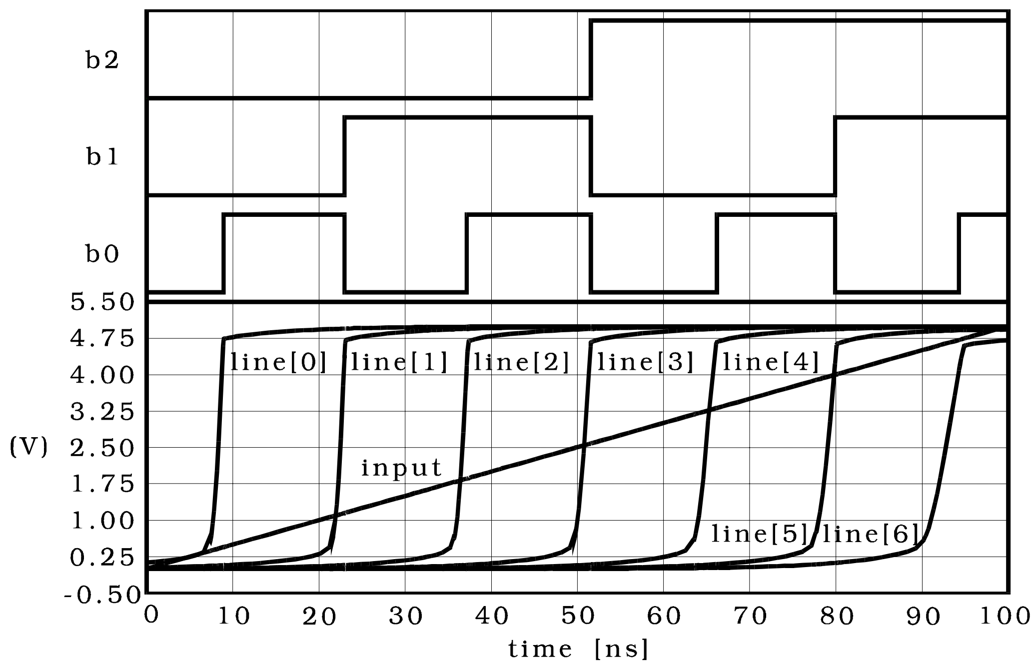


Figure 8.2. Simulation results of parallel A/D converter in CMOS technology.

Appendix 1
Alecsis installation and use

A1.1. Alecsis installation

Alecsis is program for UNIX operating system. Up to now, it was installed on following workstations:

- IBM RISC (AIX operating system),
- HP 9000s300/400 (HPUX operating system),
- HP9000s700/800 (HPUX operating system),
- Silicon Graphics (IRIX operating system),
- SUN Sparc (SUNOS operating system).

Installation is performed also for IBM PC (LINUX operating system) but is still considered shaky.

It is delivered on a single floppy disk, as a single file `alecsis.tar.Z`. File is compressed using UNIX `compress` command, and archived using UNIX `tar` command. When the archive is opened, directory `alecsis` is created, with following subdirectories:

- `src2.3` - source code;
- `include` - standard header files;
- `sys` - standard libraries (in AleC++ code);
- `lib` - standard libraries (compiled into Alecsis object code);

- `bin` - executables;
- `agnu1.1` - waveform display program (explained in separate Appendix);
- `alm` - Alecsis Library Manager (explained in separate Appendix);
- `nrl` - programs for postprocessing of digital simulation results (explained in separate Appendix);
- `p2a` - PSpice2Alecsis converter (explained in separate Appendix).

A1.1.1. Paths for UNIX shell

It is necessary to make Alecsis executable, Alecsis include files and Alecsis libraries visible from your working directory. The most convenient way is to define these paths in your UNIX shell. If you are using C-shell, you can modify the `.cshrc` file in your home directory. This is to be done before compiling Alecsis source code, as Alecsis `Makefile` uses some shell variables described here.

If Alecsis executable file `alec` is stored in directory `alecsis/bin` (which is default), you should add in your `.cshrc` file the following line:

```
set path = ($path $HOME/alecsis/bin)
```

assuming that directory `alecsis` is unpacked below your home directory `$HOME`. If you have stored executable file `alec` in some other directory, the appropriate path has to be defined.

It is also necessary to put command:

```
setenv ALEC_HOME $HOME/alecsis
```

in your `.cshrc` file. This makes standard header files located in `$HOME/alecsis/include`, and standard libraries located in `$HOME/alecsis/lib` visible. Header files, containing declarations, should have extension '`.h`' (like in C/C++). Header files, included using '<' and '>' as parentheses, would be searched for in directory `$HOME/alecsis/include`, e.g.

```
# include <alec.h>
```

List of directories where Alecsis search for included files can be expanded from the command line, using option '`-I`', explained later in this Chapter. If the file name is included using quotation marks, e.g.:

```
# include "header.h"
```

it will be searched for in current working directory.

Libraries contain definitions of modules and functions that are declared in header files. They are compiled into Alecsis object code, and have file extension '`.ao`'. Libraries included using `library` command (explained later in this Chapter) or '`-l`' command option (also explained later in this Chapter) are searched for in directory `$HOME/alecsis/lib`. Users often define their own libraries, and it is often necessary to search some other directories, too. These directories can be added using command option '-L' and shell variable ALEC_LIB_PATH. For instance:

```
setenv ALEC_LIB_PATH .:./lib:
```

specifies that definitions are searched for in: current directory '`.`'; directory '`./lib`'; and directory '`/usr/cad/alecsis/ttl`'. Colon '`:`' is used as a separator. In this example, you can see that the library

directory can be specified using relative or absolute path. You should at least specify current directory '.' using ALEC_LIB_PATH. So, you have to put into your .cshrc file:

```
set path = ($path $HOME/alecsis/bin)
setenv ALEC_HOME $HOME/alecsis
setenv ALEC_LIB_PATH .
```

A1.1.2. Compiling Alecsis source code

To compile Alecsis source code, go to directory src2.3 and type make. The Makefile will give you info about compilation on different hardware workstations. The Makefile can be modified for installations on workstations that are not on the list given above, too. Different flags are explained in the Makefile. However, we it can happen that some interventions in the source code are necessary for installation on different hardware platforms.

There is one part of the Makefile that should be edited in any case. It regards paths to the C libraries, location where executable files are stored, etc.

A1.1.3. Compiling Alecsis standard libraries

Alecsis standard libraries are given in directory alecsis/lib as compiled file -- Alecsis object code binaries (extension '.ao'). They are also given in AleC++ source code in directory alecsis/sys (extension '.ac' or '.hi'). AleC++ compiler works in the same manner on any workstation, but there might be some differences in the way data are stored on different workstations. For that reason, after Alecsis installation, libraries ought to be recompiled. Libraries are compiled using options '-c' and '-O':

```
alec -c -O file_name
```

Option '-c' means that files compiled (not interpreted), so that the file with object code is created. Option '-O' turns the optimizer on. For instance, file alloc.hi is compiled using:

```
alec -c -O alloc
```

which creates library alloc.ao. Libraries (files with extension '.ao') must be then moved to alecsis/lib directory.

The most important Alecsis standard libraries are explained in separate Appendix.

Note: For library management, special program alm (Alecsis Library Manager) is created. It is explained in separate Appendix.

A1.2. Alecsis use

Many aspect of Alecsis usage are already explained in this Manual. We will give here overview of Alecsis command line options, list of file name extensions, and some options for including precompiled libraies.

A1.2.1. Program call from the command line -- command options

The name of Alecsis executable file is **alec**. The program is invoked from UNIX command line by listing the name (alec), one or more input files, and desired options. There can be more input files on the command lines, but only one of them can be in the source (AleC++) code. Other must be object files, that are already compiled into Alecsis object code (file extension '.ao').

Alecsis normally creates file with extension '.ar' (Alecsis results), that contain results of the simulation. If it is invoked with '-c' option, it creates object files instead (extension '.ao'). That means, only compilation of AleC++ code is executed, not the interpretation of the compiled code.

Object files are similar to Alecsis libraries (extension '.aa'), which also contain compiled Alecsis object code. The only difference is in processing of their content. **See Appendix on Alecsis Library Manager (alm) on how to create and manage libraries.** Libraries have a symbol table of contents at the beginning enabling fast search. If the library is appended using option '-l', the desired entity from it (module, function, etc.) will be loaded to memory only if referenced as a global signal (in the linking phase). However, object files listed as arguments on the command line appear **in full** in the memory (not selectively). If the object file contains the **main** function (C/C++ -like), Alecsis can execute it. That means Alecsis can interpret previously compiled files - no source code in AleC++ (not compiled) is necessary. However, root module cannot be compiled and placed in libraries.

Alecsis 2.3 supports the following options (in alphabetical order):

-a listing of library content (no compilation or execution).

-c compilation of the original file (without execution). The compilation of file with name *name*.ac (or *name*.hi) will produce a file named *name*.ao.

-cl multiple searching of libraries (cycle library). The order of libraries listed is not important with this options since linker would search again in case of an undesired outcome.

-D*symbol*<=*token*> equivalent to the command in the code
"#define symbol <token>".

-E Alecsis executes the preprocessing phase only and prints the result in stdout.

-g appending of object library containing information for location of errors (in compiling), and giving the information on names and number of lines of the user code with the fatal error during the simulation, instead of the system announcement of type "bus error" or "segmentation fault". If able, simulator will give the content of the working stack at the time of fatal error (the order of function calls). Useful for debugging.

Note: Option '-g' helps in debugging AleC++ code used in processes. Errors that appear as a result of inconsistent system of equations, for instance, cannot be debugged in this way.

-i no development of inline function. All inline functions are compiled as non-inline and static (this allows the existence of functions with the same name and the same parametric profile in another library).

-I*dir* expansion of the list of directories where the library appended using include command can be found.

-l*lib* appending the library *lib*.ao to the linker list. During the resolution of unresolved external symbols, linker will successively search the libraries for these symbols. You can list desired number of libraries using this option more times. The order in the list is important since linker does not return to an already-searched library (this is important if

one external symbol refers to another unresolved external symbols). Option '-cl' (cyclic search of libraries) solves this problem.
Option '-l' is equivalent to command library in the AleC++ code.

-L*dir* expansion of the list of directories where the library can be found. This option is equivalent to setting environment variable ALEC_LIB_PATH.

-o *file*.ao the result of compilation is placed in the file listed after the option. If the file contains the function main, the command "alec *file*.ao" can execute it.

-O code optimization. Beside reducing the number of instructions, this instruction causes a number of useful warnings of variable masking, absence of function prototypes, etc.

-over ban on operator overload.

-r no recursion. It is recommended for non-recursive functions because it gives faster code.

-S creation of an assembler file *name*.as from the given source file *name*.ac.

-stat printing of the separate output library (*name*.stat) containing data on the activity of digital circuit part during the simulation (the number of events and the number of processes in every moment).

-t printing of the duration in CPU seconds for particular phases of the program (compilation, linking, preparation and execution of the simulation).

-v*verbose_level* Gives more information about the simulation run. There are following options:

- -v1 tracks symbol table activity
- -v2 tracks intermediate code generation (operand types etc.)
- -v3 all LEX tokens are printed out as they arrive
- -v4 follows voltage generator/inductor loops detection
- -v5 prints instructions as they are flushed
- -v6 tracks overloading and prototype mangling
- -v7 prints list of nodes
- -v8 follows the use of weights if option dcon is used
- -v9 follows the process of static/global initialization
- -v10 follows library management
- -v11 follows function declaration
- -v12 clear global symbol table before simulation
- -v13 tracks function prototype existence
- -v15 tracks class member access control
- -v16 follows function inline expansion
- -v31 prints system matrix and right-hand side vector in every iteration, as without reordering
- -v32 prints system matrix and rhs vector in every iteration as reordered.
- -v33 prints **both** non-reordered and reordered system matrix and rhs vector, respectively, in every iteration
- -v55 turns on full logic initialization
- -v99 changes all calls to exit() with abort() to dump core file

Note: Verbose level 55 (full logic initialization) is rather a simulation option than a verbose level, and it will be organized as such in following versions of Alecsis.

Most of these verbose levels are of interest only for us that created Alecsis, for our debugging purposes. However, there are some of them that can be very useful for Alecsis users. For instance, **verbose level 8 follows use of weight when option dcon for difficult convergence problems is used**. This can be very useful for setting correct values for options max_weight, min_weight, p, q, and maxdcon, if you are not satisfied with their default values (see Chapter 5, section on simulation options for details).

Verbose level `31` prints out system matrix, which can sometimes be helpful if you have problems with zero pivot (singular matrix). This is, however, useful only for small matrices, as it is very difficult to analyze large matrices.

Note: If more than one *verbose_level* is given, only the last one will take effect. For example:

```
alec -v1 -v2 file_name
```

has the same effect as:

```
alec -v2 file_name
```

A1.2.2. File name extensions

List of file name extensions is given in Chapter 2, but is given here again for completeness. The names of Alecsis input files are arbitrary (the maximal name length is determined by the specific operating system,) but they have to have the extension `.ac`. Extension `.hi` is also allowed for compatibility with version 1.0. File name extensions are the following:

`ac` - Alecsis input file

`hi` - Alecsis 1.0 input file (accepted by newer versions, too)

`h` - Alecsis header file (as in C/C++)

`ar` - Alecsis results (Alecsis output file, Agnu input file)

`ao` - Alecsis object-code file (compiled input file)

`as` - Alecsis assembly language file (created by compiler using option `-S`)

`aa` - Alecsis library

`stat` - Alecsis statistics file (creted when command option `-stat` is used)

A1.2.3. Listing of libraries in the source file -- command library

Beside from the command line (option '`-l`'), you can list the appended libraries in the source file, as well. The keyword used for that is **`library`**. You can list an unlimited number of libraries separated by a coma, ending with semicolon:

```
library lib1, lib2, lib3, "lib4";
```

The command can appear many times in the text, while the result is a union of all separate lists. The command can be anywhere in the text if on global level (outside functions, modules, etc.). The order of listing of libraries is important unless you use the option '`-cl`'. You can use this command in conjunction with option '`-l`'. Libraries are used in the linking phase, so command `library` should be used in the file where your `main` function or the `root module` is.

The library name can be in quotation marks (`"lib4"` in example above) if the name of the library is masked by some other name in the present context.

A1.3. Overview of Alecsis versions

We use notation of Alecsis versions with tree numbers. First number denotes crucial change of Alecsis/AleC++ functionality. The second one denotes change of functionality (new feature) from the user point of view. The last number is denotes improvement (usually debugging) of existing functions.

Alecsis 1.x	- input language based on C, no object-orientation.
Alecsis 2.1.1. - 2.1.50	- object-oriented input language AleC++ is introduced
Alecsis 2.2.1. - 2.2.33.	- operator `d2dt2` is introduced
Alecsis 2.3.1. - 2.3.x	- through and across `eqn` statements are introduced

Appendix 2
Alecsis standard libraries

In this Appendix, only headers of standard libraries. These headers are included using included command. In headers, all declarations are given. All definitions (modules, functions, etc.) mentioned in the headers are in one file. For instance, header `alec.h` corresponds to library `alec.ao`, referenced as library `alec`.

Only some of the files that are stored in directory `alecsis/include` are given here. In future issues of this Manual, we will give the most important headers specific for digital simulation, too.

Files are given in alphabetical order:

A2.1 alec.h

This is a standard library containing all main definitions and declarations needed. The header is in the standard place (directory alecsis/include) under the name `alec.h`, and the library body is in the file `alec.ao` (directory `alecsis/lib`). Header file heeds to be included using `include` command. Library `alec` need not to be given explicitely using '`-l`' option or `library` command, as it is standard library.

The content of the header file is following:

```
/* Faculty of Electronic Engineering Nis
 * Alecsis 2 hybrid simulator library header file
 * Library: alec
 *
 * These declarations are not a must, but compiler will complain
```

```
 * about parameters for functions listed below.
 *
 */
#ifndef _ALEC_INCLUDED
#  define _ALEC_INCLUDED

#  ifndef NULL
#    define NULL  (0)
#  endif /* NULL */

#  ifndef EOF
#    define EOF   (-1)
#  endif /* EOF */

# ifndef MAX
#  define MAX(_x, _y) ((_x)>(_y) ? (_x) : (_y))
# endif /* MAX */

# ifndef MIN
#  define MIN(_x, _y) ((_x)<(_y) ? (_x) : (_y))
# endif /* MIN */

# define cout stdout
# define cin  stdin
# define cerr stderr

   typedef struct {
 int   __cnt;
 char  *__ptr;
 char  *__base;
 int   __flag;
 char  __file;
   } FILE;

   /* standard built-in function prototypes */

   extern  int printf(const char *, ... );
   extern  int fprintf(FILE *, const char *, ...);
   extern  int sprintf(char *, const char *, ...);
   extern  int fputc(char, FILE *);
   extern  int putc(char, FILE *);
   extern  int putchar(char);
   extern  char fgetc(FILE *);
   extern  char getc(FILE *);
   extern  char *gets(char *);
   extern  char *fgets(char *, int, FILE*);
   extern  char getchar(void);
   extern  int exit(int=0);
   extern  FILE *fopen(const char *, const char *);
   extern  int fclose(FILE *);
   extern  int fflush(FILE *);
   extern  int feof(FILE *);
   extern  int fseek(FILE *, int, int);
   extern  int ftell(FILE *);
   extern  int rewind(FILE *);
   extern  int fwrite(const void *, int, int, FILE *);
   extern  int fread(void *, int, int, FILE *);
   extern  void *calloc(int, int);
   extern  void *malloc(int);
   extern  void free(void *);
   extern  double node_value(int, int);
   extern  double drand(void);
   extern  void srand(int=1);
```

```
   extern  double time_now(void);
   extern  void warning (const char *, int=0);
   extern  char *strcpy(char *, const char *);
   extern  int strcmp(const char *, const char *);
   extern  int strlen(const char *);
   extern  int get_info (int);
   extern  int get_info (int, char *);
   extern  int atoi(const char*);
   extern  double atof(const char*);
   extern  int system(const char*);

   extern int set_bpoint(double);

 /* standard Boolean choices */
# define True  1
# define False 0

 /* alternative solution - Bool type */
typedef enum { false, true } Bool;

 /* switch-like choices */
# define On  1
# define Off 0

 /* Integration method choices */
# define None           0
# define EulerBackward  1
# define Gear2          2

        /* Matrix renumeration options */
// # define None  0
# define Fast    1
# define Best    2
# define Frontal 3

 /* dcon choices */
# define Initial 1
# define Always  2

 /* get_info selection indx */
# define CurrentPath   0
# define ParentPath    1
# define CurrentLevel  2

 /* the most common implicit aliases */
implicit {
 resistor   r;
 capacitor  c;
 inductor   l;
 vgen       v;
 cgen       i;
 mosfet     m;
 bjt        q;
 jfet       j;
 diode      d;
 switch     s;
};

 /* pulse generator defined as a module */

module pulse (n1, n2) action (    double vlo,      // low level
                 double vhi,      // high level
                 double tr,       // rise delay
```

```
                    double twl,      // high level pulse width
                    double tf=0,     // fall delay
                    double twh=0,    // low level pulse width
                    double td=0      // start delay time
                   );

# ifndef _BIT_INCLUDED
#   include <bit.h>
# endif /* _BIT_INCLUDED */

 /* overloaded << and >> - C++ style I/O */

FILE *operator<< (FILE *fp, int i);
FILE *operator<< (FILE *fp, double d);
FILE *operator<< (FILE *fp, const char *s);
FILE *operator<< (FILE *fp, char c);

FILE *operator>> (FILE *fp, char &c);
#endif /* _ALEC_INCLUDED */
```

This library contains prototypes of all intrinsic functions of general usage. The rest of C functions (except the mathematical) are **not implemented** and cannot be called.

Command `implicit`, used in this file, gives possibility to use Alecsis similarly as SPICE.

We realized trapezoidal voltage generator as a `module` - its declaration is given above.

Beside standard C functions, `alec.h` contains a prototype of function **`warning`**, that is specific for simulaiton, and is therefore not a standard C-function. You can use this function to print information of place, time and context (the process, current component, etc.) during the execution. If the argument differs from 0, the simulator stops the execution, otherwise the simulation continues.

A2.2. ctype.h

This is not a real library since it has only the header file - there is no file named `ctype.ao`. This library contains the definitions of macros for work with characters:

```
#ifndef _CTYPE_INCLUDED
# define _CTYPE_INCLUDED

#  define _U 01
#  define _L 02
#  define _N 04
#  define _S 010
#  define _P 020
#  define _C 040
#  define _B 0100

     extern const char __ctype[];
     extern const char __upshift[];
     extern const char __downshift[];

#  define isalpha(__c)  (__ctype[__c]&(_U|_L))
#  define isupper(__c)  (__ctype[__c]&_U)
#  define islower(__c)  (__ctype[__c]&_L)
#  define isdigit(__c)  (__ctype[__c]&_N)
#  define isalnum(__c)  (__ctype[__c]&(_U|_L|_N))
#  define isspace(__c)  (__ctype[__c]&_S)
```

```
#  define ispunct(__c)  (__ctype[__c]&_P)
#  define isprint(__c)  (__ctype[__c]&(_P|_U|_L|_N|_B))
#  define isgraph(__c)  (__ctype[__c]&(_P|_U|_L|_N))
#  define iscntrl(__c)  (__ctype[__c]&_C)

#  define isascii(__c)  ((__c) <= 0177)
#  define toupper(__c)  ((__upshift)[__c])
#  define tolower(__c)  ((__downshift)[__c])
#  define toascii(__c)  ((__c))

#endif
```

A2.3. bit.h

This file contains only declarations, **while the definitions are in the library `alec`**. This is the elementary system of state `bit` containing `'0'` and `'1'` - logic zero and one, as well as the functions for overload of elementary operators:

```
/*
 * Faculty of Electronic Engineering Nis
 * Alecsis 2.0 hybrid simulator library header file
 * Library: alec
 * Content: predefined two-valued logic system
 */

#ifndef _BIT_INCLUDED
# define _BIT_INCLUDED

 typedef enum { '0', '1', ' '=void, '_' = void } bit;

 /* overloaded logical operators for type "bit" */
 extern bit operator~ (bit);
 extern bit operator& (bit, bit);
 extern bit operator| (bit, bit);
 extern bit operator^ (bit, bit);
 extern bit operator~& (bit, bit);
 extern bit operator~| (bit, bit);
#endif
```

A2.4. gnulib.h

Functions declared here are used for on-line viewing of simulation results. They enable inter-process communication with program `gnuplot`, that is used for viewing of simuation results. Function bodies are in library `gnulib`, that is to be appended using library command or '`-l`' command option.

```
#ifndef _GNULIB_INCLUDED
# define _GNULIB_INCLUDED
#include <alec.h>
#include <unistd.h>

typedef enum { Lines, Points, LinesPoints, Impulses } LineStyle;

struct TraceList {
 char *name;
 int channel;
 LineStyle lst;
```

```
  struct TraceList *next;
};
typedef enum { CallGnu, Close } ExitStyle;


class gnu {
   protected:
  char *gnufile;
  char *title;
  FILE *fp;
  int ntraces, nchannels;
  LineStyle line_style;
  TraceList *traceh, *tracet;
    public:
  gnu (const char* ="gnu.dat", const char* = "Alecsis results");
  ~gnu();
  set_line_style(LineStyle);
  add_trace(const char*, LineStyle = Lines, int = 1);
  add_traces (const char*, ...);
  add_data(double, Bool);
  update  (double, ...);
  close_data ( ExitStyle, const char *geom="" );
  report();
};

struct ResultBuffer {
  double time;
  double *values;
  struct ResultBuffer *next;
};

class GnuOnLine: public gnu {
   protected:
  double tstop, update_period;
  double tprint, last_time;
  double vmin, vmax;
  double cmin, cmax;
  Bool channel_open, first_update;
  int channel[2];
  ResultBuffer *rbh, *rbt;
  char *plot_mess;
  create_plot_mess();
  initialize();
  flush_results();
  tell_gnu(const char*);
  set_value_margins();
   public:
  GnuOnLine (const char* ="gnu.dat",
                  const char* = "Alecsis results");
  ~GnuOnLine ();
  time_frame(double, double, double=0.0);
  value_frame (double, double);
  open_channel (const char* ="");
  open_channel (double, double, double, double,
                        double = 0.0, const char* = "");
  send_data (double, ...);
  main_loop();
};
#endif
```

The most appropriate way to explain these functions is to give an example. Here is an ring oscillator description, where CMOS inverter is used as ring element. This inverter is cloned using `clone` command in an ring of inverters (5 inverters in our example).

```
/*************************************************************/
spice {
# include "omos.mod"
}
# include <alec.h>
#include <gnulib.h>
library gnulib;
/*************************************************************/
    /* define analog inverter as a subcircuit */

module inverter ( In, Out, vdd, vss ) {
  mup (Out, In, vdd, vdd){ model=pomos;l=5u; w=35u;ad=as=170p;
                          pd=ps=50u;};
  mdwn(Out, In, vss, vss){ model=nomos;l=5u; w=10u;ad=as=50p;
                          pd=ps=35u;};
}
/*************************************************************/
module ring (node vdd, vss; node nodes[]) {
 inverter inv;

 action structural (int nring) {
    int i;
    for (i=0; i<nring;  i++) {
       if (i==nring-1)
        clone inv [i] (nodes[0], nodes[i], vdd, vss);
       else
        clone inv [i] (nodes[i+1], nodes[i], vdd, vss);
    }
 }
}
/*************************************************************/
#define Period 50ns

root module ring_oscilator () {
 ring rn;
 vgen vdd;
 node tmp[5] <- { 0.5v, 4.5v, 0.5v, 4.5v, 0.5v };

 vdd (Vdd, 0) 5v;
 rn (Vdd, 0, tmp) action (5);

 timing { tstop = Period; a_step = a_stepmin = Period/200; }

 out { node tmp[0];
       node tmp[1];
       node tmp[2];
       node tmp[3];
       node tmp[4]; }

#ifdef GNU
 action {
    static GnuOnLine gp ("ring", "Ring oscillator");
    process initial {
       gp.add_traces("tmp0", "tmp1", "tmp2", "tmp3", "tmp4", 0);
       gp.time_frame(Period, Period/100);
       gp.value_frame(0v, 5v);
       gp.open_channel();
    }
    process post_moment {
       gp.send_data(tmp[0], tmp[1], tmp[2], tmp[3], tmp[4]);
    }
    process final {
       gp.main_loop();
```

```
        }
    }
#endif
}
/**************************************************************/
```

Usage of gnulib functions are under preprocessor option

```
# ifdef GNU
```

so they will be activated if Alecsis is invoked using:

```
alec -DGNU ring
```

where `ring.ac` is the name of the file listed above.

Firstly, an instance of class `GnuOnLine` with name `gp` is declared. Constructor receives two parameters (both have default values). First is the file name, used to store waveform data, in this example `ring.dat` (extension '`.dat`' is added to the given name). The second parameter is waveform title, to appear above the graphics.

Class method `add_traces` is invoked in `process initial`. It gives the number of waveforms to be traced, and their names. This method is realized as the function with variable name of arguments, and the list of arguments has to be finished with `0`. All waveforms are drawn on one graphics, i.e. with one y-axis.

Method `time_frame` gives the last time point for x-axis (the first time point is assumed to be `0`), and the time step for waveform updating. Method `value_frame` gives **initial** range for y-axis scale (min and max). It is to be noted that y-axis range will be automatically updated during the simulation, as we usually do not know range of our simulation results in advance. However, x-axis cannot be updated, so arguments to `time_frame` must be correct.

Method `open_channel` opens the communication channel with `gnuplot`.

In the `process post_moment`, method `send_data` is used. It is invoked as `post_moment`, since simulation results can be sent **after** given time instant is solved. The number of arguments has to agree with number of non-zero arguments of `add_traces`.

In the process final, function `main_loop` is invoked. Without this function, `gnuplot` will close the drawing when the simulation is finished. Function `main_loop` leave program `gnuplot` active and gives you normal `gnuplot` prompt, so you can analyse the waveforms, create different output formats, etc., when the simulation is finished.

Class `GnuOnLine` can be used for all standard simulation problems. However, we have used Alecsis to solve partial differential equations (simulation of micromechanical sensors), and have also created on-line viewing for spatial 3D drawings. Class `GnuOnLine` was used there as a *base class* to create *derived classes* for specific problems.

A2.5. math.h

We implemented all mathematics functions as instructions of virtual processor of simulation, so their execution does not require including this header nor appending of a library body. The following declarations are given so you can see what was implemented, and what was not:

```
/*****************************************************************
 * This file contains prototype declarations for ALECSIS2.0
 * built-in math functions. Since they really work as
 * instructions, the declarations are necessary just that compiler
```

```
 * does not complain about arguments when invoked with -O
 * (optimizer) option. The simulator works O.K. without this file.
 ***************************************************************/
#ifndef MATH_INCLUDED
#  define MATH_INCLUDED

     extern double acos(double);
     extern double asin(double);
     extern double atan(double);
     extern double atan2(double, double);
     extern double cos(double);
     extern double sin(double);
     extern double tan(double);
     extern double cosh(double);
     extern double sinh(double);
     extern double tanh(double);
     extern double exp(double);
     extern double log(double);
     extern double log10(double);
     extern double pow(double, double);
     extern double sqrt(double);
     extern double ceil(double);
     extern double fabs(double);
     extern double floor(double);

     extern int abs(int);

// These constants may be of some help in modelling

#  define M_E           2.7182818284590452354
#  define M_LOG2E       1.4426950408889634074
#  define M_LOG10E      0.43429448190325182765
#  define M_LN2         0.69314718055994530942
#  define M_LN10        2.30258509299404568402
#  define M_PI          3.14159265358979323846
#  define M_PI_2        1.57079632679489661923
#  define M_PI_4        0.78539816339744830962
#  define M_1_PI        0.31830988618379067154
#  define M_2_PI        0.63661977236758134308
#  define M_2_SQRTPI    1.12837916709551257390
#  define M_SQRT2       1.41421356237309504880
#  define M_SQRT1_2     0.70710678118654752440
#endif
```

A2.6. unistd.h

An equivalent of UNIX `unistd.h` file. There is no appropriate library. Here are declarations of functions for forking and pipelining, used to enable simultaneous simulation and viewing of results (see section on `gnulib.h`).

```
#ifndef _UNISTD_INCLUDED
# define _UNISTD_INCLUDED

/*
 * unistd.h
 * symbolic constants and structures which are used
 * for support of the /usr/group standard.
 *
 */
```

```
#  ifndef NULL
#    define NULL  0
#  endif

#  ifndef R_OK
/* Symbolic constants for the "access" function: */
#    define R_OK  4
#    define W_OK  2
#    define X_OK  1
#    define F_OK  0
#  endif

/* Symbolic constants for the "lseek" function: */
#  ifndef SEEK_SET
#    define SEEK_SET   0
#    define SEEK_CUR   1
#    define SEEK_END   2
#  endif
#  include <time.h>

       extern int close(int);
       extern int dup(int);
       extern int execl(const char *, const char *, ...);
       extern int execv(const char *, const char **);
       extern int pipe(int *);
       extern int read(int, char *, int);
       extern int write(int, const char *, int);
       extern int fork();
       extern int _wait();
       extern int select(int, int*, int*, int*, struct timeval*);

#endif
```

A2.7. varargs.h

Two macros for work with functions and `action` blocks with variable number of arguments (no library body) are in the file `varargs.h`. Explanations of how to use them are in Chapter 4 for functions, and in Chapter 5 for `action` parameters. Meaning of definition of `DWORD_ALIGNMENT` is explained there, too.

```
#ifndef VARARGS_INCLUDED
#  define VARARGS_INCLUDED
//     ALECSIS2.0 Header file
//     (from standard C "varargs.h")

// Use __mode int for char and short types.

# define TYPE_DOUBLE 8
# define DWORD_SIZE  8

# define va_start(__list, __par) (__list = ((char *) (&__par) +
sizeof(__par)))
# ifndef DWORD_ALIGNMENT
#   define va_arg(__list,__mode) ((__mode *)(__list += \
                                       sizeof(__mode)))[-1]
# else  /* DWORD_ALIGNMENT */
#  define va_arg(__list,__mode) ((sizeof(__mode)==TYPE_DOUBLE && \
                                     ((int)__list%DWORD_SIZE) ) ? \
                              ((__mode *)(__list+=sizeof(__mode)+ \
                          (DWORD_SIZE-((int)__list%DWORD_SIZE)))) \
               : ( (__mode *)(__list += sizeof(__mode)) )  )[-1]
```

```
# endif /* DWORD_ALIGNMENT */

# endif /* VARARGS_INCLUDED */
```

Appendix 3
Syntax of AleC++

We systematized the syntax of AleC++ in the following text.

Operator	Association
[] () . -> ::	left
~ ! sizeof lengthof - ++ -- (cast) new delete $ @	right
.* ->*	left
* / %	left
+ -	left
<< >>	left
< <= > >=	left
== !=	left
& ~&	left

^ ~^	left
| ~|	left
?:	right
= <- *op*=	right
,	left

Syntax begins from the symbol **global_data**:

```
/***************************************************************/
global_data:
          global_item
          global_data global_item

global_item:
          external_definition
          module_definition
          implicit_definition
          model_card
          spice_code
          library_specification

/********************** #1.1 Function definition **********************/
nonempty_fpar_list:
          on_line_declaration

on_line_declaration:
          on_line_decl
          on_line_decl ,  ...
          ...

on_line_decl:
          on_line_item
          on_line_decl , on_line_item

on_line_item:
          auto_decl_specifiers   on_line_declarator

on_line_declarator:
          init_declarator
          abstract_declarator   initializer

function_body:
          function_statement

function_statement:
          compound_statement
```

/*********************** #1.2 Data definition *************************/
external_definition:
    <decl_specifiers> externals
    decl_specifiers

externals:
    declarator ctor_initializer function_body
    declaration_list

/******************** Base constructor initializer *************************/
ctor_initializer:
    : member_initializer_list
    : (expression_list)

member_initializer_list:
    member_initializer
    member_initializer_list , member_initializer

member_initializer:
    class_name (expression_list)
    identifier (expression_list)

/********************** #1.3 Module definition *************************/
module_definition:
    module module_definition_or_decl
    root_module_definition

root_module_definition:
    root <**module**> module_prototype module_body

module_definition_or_decl:
    module_prototype module_body
    module_prototype_list ;

module_prototype_list:
    module_prototype_item
    module_prototype_list , module_prototype_item

module_prototype:
    <signal_sc> full_module_name < (<module_interface>) >

full_module_name:
    module_name
    class_name :: module_name

module_name:
    <any_name.>module_id

module_id:
    identifier
    class_name

module_prototype_item:
    module_prototype <actpars>

actpars:
    action *(<fpar_list>)*

module_interface:
    partial_interface
    full_inter_decl
    vararg_interface
    full_inter_decl ; vararg_interface

partial_interface:
    partial_node
    partial_interface , partial_node

partial_node:
    free_form_node

full_inter_decl:
    full_interface
    full_inter_decl ; full_interface

full_interface:
    arbit_sc <signal_type_specifiers> <direction> full_inter_list

signal_type_specifiers:
    auto_decl_specifier
    signal_type_specifiers auto_decl_specifier

vararg_interface:
    arbit_sc <signal_type_specifiers> <direction> ...

arbit_sc:
    signal_sc
    type_specifier

full_inter_list:
    signal_init_declarator
    full_inter_list , signal_init_declarator

signal_declarator:
    free_form_node
    signal_declarator [array_size]
    signal_declarator [**auto** *]*

signal_init_declarator:
signal_declarator <initializer> converters

signal_sc:
signal
node
charge
current
flow

direction:
in
out
inout

converters:
: (<module_name , module_name)
: module_name

free_form_node:
identifier
integer_constant

/********************** #1.3.1 Module body ***************************/
module_body:
{ <structural_decl> <component_map>
<conversion_spec> <simulation_spec> <action_decl> }

/********************** #1.3.1.1 Structural declaration *******************/
structural_decl:
local_declaration
structural_decl local_declaration

local_declaration:
signal_declaration
component_declaration

signal_declaration:
arbit_sc <signal_type_specifiers> signal_list ;

signal_list:
signal_init_declarator
signal_list , signal_init_declarator

component_declaration:
component_sc component_list ;

component_list:
one_component
component_list , one_component

component_declarator:
    identifier
    component_declarator [array_size]

one_component:
    component_declarator

component_sc:
    module module_prototype_item
    module_name
    builtin_element_type
    switch

full_name:
    identifier
    library_name . identifier

/******************** #1.3.1.2 Component mapping *********************/
component_map_list:
    component_map
    component_map_list component_map

component_map:
    component_name (<actual_signal_list>) parameters

component_name:
    any_name
    return module_name

parameters:
    ;
    constant_expression ;
    par_assignment ;
    special_assignment <special_assignment> ;
    { parameter_list } <;>
    { pwl_list <;> } <;>

special_assignment:
    model_attach
    action_positional

action_positional:
    action (<expression_list >)

actual_signal_list:
    actual_signal
    actual_signal_list , actual_signal

actual_signal:
    static_signal
    void

pwl_list:
    pwl_pair
    pwl_list ; pwl_pair

pwl_pair:
    assignment_expression , assignment_expression

/******************* #1.3.1.3 Conversion declaration *****************/
conversion_spec:
    conversion *{ parameter_list } <;>*

/*************** #1.3.1.4 Simulation conditions (root module only) *********/
simulation_spec:
    simulation_item
    simulation_spec simulation_item

simulation_item:
    options_list
    timing_list
    output_list

options_list:
    options *{ parameter_list } <;>*

timing_list:
    timing *{ parameter_list } <;>*

output_list:
    out *{ output_groups } <;>*

output_groups:
    output_group
    output_groups output_group

output_group:
    arbit_sc <signal_type_specifiers> <direction> output_item_list ;
    caption *string_constant ;*
    sweep *arbit_sc <signal_type_specifiers> output_item ;*

output_item_list:
    output_item
    output_item_list , output_item

output_item:
    <path_sc /> static_signal <conv_select>
    identifier compound_statement

path_sc:
    path_level
    path_sc / path_level

path_level:
    identifier
    string_constant

conv_select:
    (*identifier*)

/********************* #1.3.1.5 Action declaration *******************/
action_decl:
    action <*update*> *generic action_body*

update:
    structural
    post_structurual
    initial
    per_moment
    post_moment
    per_iteration
    final

generic :
    (<*fpar_list*>)

action_body:
    { <*action_context*> }

action_context:
    action_statements

action_statements:
    action_statement
    action_statements action_statement

action_statement:
    statement
    process_statement

process_statement:
    process_header process_update compound_statement

process_header:
    <*identifier :*> **process**

process_update:
    <**virtual**> *update*

(sensitivity_list)

sensitivity_list: sensitive_signal
sensitivity_list , sensitive_signal
sensitivity_list ...

sensitive_signal:
static_signal

static_signal:
free_form_node
$ constant_expression
:: identifier
(static_signal)
static_signal [constant_expression]
static_signal [constant_expression : <constant_expression>]
static_signal . identifier

*/********************** #1.5 Implicit definition ************************/*
implicit_definition:
implicit *{ implicit_context } <;>*

implicit_context:
implicit_list
implicit_context implicit_list

implicit_list:
component_sc short_list ;

short_list:
identifier
short_list , identifier

*/********************** #1.6 Library specification ********************/*
library_specification:
library *library_list ;*

library_list:
library_item
library_list , library_item

library_item:
any_name

*/********************** #2.1 Declarations ***************************/*
declaration:
decl_specifiers <init_declarator_list>;

decl_specifiers:
decl_spec_list

decl_spec_list:
first_decl_specifier
decl_spec_list decl_specifier

auto_decl_specifiers:
auto_decl_spec_list

auto_decl_spec_list:
type_specifier
auto_decl_spec_list auto_decl_specifier

decl_specifier:
first_decl_specifier
bus_resolution_specifier
attribute_specifier

first_decl_specifier:
type_specifier
sc_specifier
signal_sc
fct_specifier
friend

auto_decl_specifier:
type_specifier
bus_resolution_specifier
attribute_specifier

length_specifier:
long
short
signed
unsigned

cv_qualifier:
const
volatile

sc_specifier:
auto
extern
static
typedef
register

fct_specifier:
inline
virtual

bus_resolution_specifier:
    : function_name

attribute_specifier:
    @ signal_attribute_sc

signal_attribute_sc:
    simple_type_name <(<expression_list >) >

type_specifier:
    simple_type_name
    struct_union_specifier
    enum_specifier
    cv_qualifier

basic_types:
    void
    char
    int
    float
    double

simple_type_name:
    basic_types
    length_specifier
    class_enum_name
    typedef_name

/********************* #2.2 Declarators *****************************/
/********************* #2.2.1 Names ********************************/
declarator_name:
    identifier
    operator_name
    qualified_name

qualified_name:
    class_name :: operator_name
    class_name :: conversion_fct_name
    class_name :: identifier

/******************** #2.2.2 Lists **********************************/
init_declarator_list:
    init_declarator
    init_declarator_list , init_declarator

/******************** #2.2.3 Items ********************************/
declarator_item:
    declarator

ptr_operator:

* <cv_qualifier>
& <cv_qualifier>
class_name :: * <cv_qualifier>
class_name :: & <cv_qualifier>

declarator:
primary_name
qualified_name
class_name
~ class_name
> class_name
ptr_operator declarator
(declarator)
declarator [array_size]
declarator (<on_line_declaration>) <cv_qualifier>

init_declarator:
declarator <initializer>

operator_name:
operator opname

opname:
(svioperatori osim $ $$, .* ->* <- ?: @)

conversion_function_name:
operator simple_type_name <ptr_operator>

/********************** #2.3 Structures/unions ***************************/
struct_union_specifier:
struct_header struct_decl_list }
struct_keyword identifier
struct_keyword tag_name
union_header union_decl_list }
union tag_name

struct_keyword:
struct
class

/********************** #2.3.1 Class/structure header *******************/
struct_header:
struct_keyword <tag_name> <base_spec> {

base_spec:
: base_list

base_list:
base_specifier

base_list , base_specifier

base_specifier:
<base_access> class_name <**virtual**>
<base_access> <**virtual**> class_name

base_access:
public
private
protected

/***********************#2.3.2 Union declaration *********************/
union_header:
union <tag_name> {

union_decl_list:
union_declaration
union_decl_list union_declaration

union_declaration:
auto_decl_specifiers union_declarator_list ;

union_declarator_list:
union_declarator
union_declarator_list , union_declarator

union_declarator:
declarator_item

struct_decl_list:
struct_declaration
struct_decl_list struct_declaration

struct_declaration:
<access_specifier> < decl_specifiers> <struct_list>;
<access_specifier> declarator <ctor_initializer> function_body <;>
<access_specifier> **friend module** friend_module_list ;

friend_module_list:
friend_module_name
friend_module_list , friend_module_name

access_specifier:
base_access :

/********************** #2.4 Initialization *************************/
initializer:
= init_expr
<- init_expr
(expression_list)

init_expr:
    constant_expression
    { }
    { initializer_list }
    { initializer_list , }

initializer_list:
    constant_expression
    initializer_list , initializer_list
    { initializer_list }

/******************* #2.5 Names for type conversion ******************/

type_specifiers:
    type_specifier
    type_specifiers type_specifier

type_name:
    type_specifiers <abstract_declarator>

restricted_type_name:
    type_specifier <restricted_declarator>

abstract_declarator:
    ptr_operator abstract_declarator
    abstract_declarator (< on_line_declaration >)
    abstract_declarator [array_size]
    (abstract_declarator)

restricted_declarator:
    ptr_operator restricted_declarator
    restricted_declarator restricted_array

restricted_array:
    [expression]

typedef_name:
    new_type (expression_list)

/******************** #2.6 Enumeration type ***********************/

enum_specifier:
    enum *<tag_name> { enum_list }*
    enum *tag_name*

enum_list:
    enumerator
    enum_list , enumerator

enumerator:

enum_symbol
enum_symbol = constant_expression
enum_symbol = **void**

enum_symbol:
identifier
character_constant

/************************** #2.7 Array size ************************/
array_size:
<constant_expression>
constant_expression : constant_expression

/************************** #3 Expressions************************/
/************************** #3.1 Literals ***************************/
constant:
integer_constant <identifier>
double_constant <identifier>
character_constant

string:
string_cat

string_cat:
string_constant
string_cat string_constant

any_name:
string_constant
identifier

asgnop:
= += -= *= /= %= &= |= ^= >>= <<=

primary_name:
identifier
operator_name

/*********************** 3.2 Primary expression ********************/
primary_expression:
basic_name
constant
string
this
:: primary_name
(expression)
now
(signal_sc) free_form_node
$$

/*********************** 3.3 Postfix expression **********************/

postfix_expression:
primary_expression
postfix_expression (<expression_list>)
postfix_expression [expression]
simple_type_name (expression_list)
postfix_expression ++
postfix_expression --
postfix_expression . basic_name
postfix_expression -> basic_name

/**********************3.4 Expression list *************************/

expression_list:
assignment_expression
expression_list , assignment_expression

/***********************3.5 Unary expression **********************/

unary_expression:
postfix_expression
++ unary_expression
-- unary_expression
* cast_expression
& cast_expression
- cast_expression
+ cast_expression
! cast_expression
~ cast_expression
sizeof (type_name)
sizeof unary_expression
lengthof unary_expression
allocator
deallocator
@ cast_expression
$ cast_expression

allocator:
<::> **new** <(expression_list)> (type_name)
<::> **new** <(expression_list)> restricted_type_name (expression_list)

deallocator:
<::> **delete** <[expression]>cast_expression

/******************* 3.6 Cast expression *************************/

cast_expression:
unary_expression
(type_name) cast_expression

/******************** 3.7 PM-expression *************************/

pm_expression:
cast_expression

```
        pm_expression  .* cast_expression
        pm_expression  ->* cast_expression

/******************* 3.8   Multiplicative expression *********************/
multiplicative_expression:
        pm_expression
        multiplicative_expression *  pm_expression
        multiplicative_expression /  pm_expression
        multiplicative_expression % pm_expression

/********************       3.9   Additive  expression **********************/
additive_expression:
        multiplicative_expression
        additive_expression  +  multiplicative_expression
        additive_expression  -  multiplicative_expression

/******************* 3.10   Shift  expression ***************************/
shift_expression:
        additive_expression
        shift_expression  <<  additive_expression
        shift_expression  >> additive_expression

/******************** 3.11   Relational  expression **********************/
relational_expression:
        shift_expression
        relational_expression  < shift_expression
        relational_expression  <= shift_expression
        relational_expression  > shift_expression
        relational_expression  >= shift_expression

/******************** 3.12   Equality  expression **********************/
equality_expression:
        relational_expression
        equality_expression  ==  relational_expression
        equality_expression  !=  relational_expression

/******************** 3.13 AND  expression **************************/
AND_expression:
        equality_expression
        AND_expression  &  equality_expression
        AND_expression ~& equality_expression

/******************** 3.14   Exclusive-OR expression *******************/
exclusive_OR_expression:
        AND_expression
        exclusive_OR_expression  ^ AND_expression
        exclusive_OR_expression  ~^ AND_expression

/******************** 3.15   Inclusive-OR  expression ******************/
inclusive_OR_expression:
```

exclusive_OR_expression
inclusive_OR_expression | exclusive_OR_expression
inclusive_OR_expression ~| exclusive_OR_expression

*/********************** 3.16 Logical AND expression ********************/*
logical_AND_expression:
inclusive_OR_expression
logical_AND_expression && inclusive_OR_expression

*/********************** 3.17 Logical OR expression **********************/*
logical_OR_expression:
logical_AND_expression
logical_OR_expression || logical_AND_expression

*/********************** 3.18 Conditional expression *********************/*
conditional_expression:
logical_OR_expression
logical_OR_expression ? expression : conditional_expression

*/********************** 3.19 Assignment expression *******************/*
assignment_expression:
conditional_expression
unary_expression asgnop assignment_expression

*/************************* 3.20 Expression ******************************/*
expression:
assignment_expression
expression , assignment_expression

*/********************** 3.22 Constant expression **********************/*
constant_expression:
conditional_expression

*/***************************#4 Statements ******************************/*
compound_statement:
{ <statement_list> }

statement_list:
statement
statement_list statement

statement:
compound_statement
simple_statement
labeled_statement
declaration
asm_statement

simple_statement:
expression ;

if_statement
while_statement
do_statement
for_statement
switch_statement
break_statement
continue_statement
return_statement
goto_statement
alecsis_statement
;

if_statement:
if *(expression) statement*
if *(expression) statement else statement*

while_statement:
while *(expression) statement*

do_statement:
do *statement* **while** *(expression) ;*

for_statement:
for *(<for_expression> <expression>; <expression>) statement ;*

for_expression:
<expression >;
<declaration>;

switch_statement:
switch *(expression) { <switch_body> }*

switch_body:
switch_groups

switch_groups:
switch_group
switch_groups switch_group

switch_group:
switch_items statement_list
default_item

switch_items:
switch_item
switch_items switch_item

switch_item:
case *constant_expression :*

default_item:
  default *: statement_list*

break_statement:
  break *;*

continue_statement:
  continue *;*

return_statement:
  return *;*
  return *expression ;*

goto_statement:
  goto *Identifier ;*

labeled_statement:
  identifier : simple_statement

asm_statement:
  asm *asm_code*

asm_code:
  asm_line ;
  { asm_lines }

asm_lines:
  asm_line
  asm_line s asm_sep asm_line

asm_sep :
  newline
  ;

asm_line:
  <identifier :> nolab_asm_line

nolab_asm_line:
  fixer asm_instr <.character_constant> <asm_operand<,asm_operand>>

fixed_instr:
  !
  volatile

asm_operand:
  logical_OR_expression
  % register_name
  (% register_name)

```
/******************** 4.1   Specific Alecsis statements *******************/
alecsis_statement:
        matrix_fillin_statement
        clone_statement
        signal_assign_statement
        wait_statement
        nlgen_statement
        allocate_statement


/******************** 4.1.1 matrix fillin statement ***********************/
matrix_fillin_statement:
        eqn  any_equation_statement

any_equation_statement:
        simple_equation_statement
        through_equation_statement
        across_equation_statement

/******************* 4.1.1.1 simple equation statement *******************/
simple_equation_statement:
        matrix_column : fillin_list = constant_expression ;

fillin_list:
        matrix_entry
        fillin_list + matrix_entry
        fillin_list - matrix_entry

matrix_entry:
        multiplicative_expression *  ddt  matrix_column_pair
        ddt  matrix_column_pair
        -  ddt  matrix_column_pair
        +  ddt  matrix_column_pair
        multiplicative_expression

ddt:
        ddt
        idt
        dtdt2

matrix_column_pair:
        { static_signal } < . identifier>
        { static_signal  , static_signal } < . identifier>

/***************** 4.1.1.2 through equation statement *******************/
through_equation_statement:
        matrix_column_pair = fillin_list ;

/****************** 4.1.1.3 across equation statement *******************/
across_equation_statement:
```

matrix_column , matrix_column_pair = fillin_list ;

/******************** 4.1.2 Signal assignment ************************/

signal_assign_statement:

postfix_expression <- <**transport**> *ass_value ;*

ass_value:

assignment_expression

delay_list

delay_list:

assignment_expression **after** *constant_expression*

delay_list , assignment_expression **after** *constant_expression*

/******************** 4.1.3 Wait statement ***************************/

wait_statement:

wait <*sensitivity_list*> *condition_clause timeout_clause ;*

condition_clause:

<**while** *expression* >

timeout_clause:

<**for** *expression*>

/******************** 4.1.4 Clone statement *************************/

clone_statement:

clone *clone_set ;*

clone_set:

one_clone_el

{ clone_list }

clone_list:

one_clone_el

clone_list one_clone_el

one_clone_el:

identifier <[constant_expression]> (<actual_signal_list>) parameters

/********************** 4.1.5 Nlgen statement ************************/

nlgen_statement:

nlgen_key identifier = conditional_expression <{ partial_derivative_list }> ;

nlgen_key:

nlcgen

nlvgen

nlgen

partial_derivative_list:
partial_derivative
partial_derivative_list partial_derivative

partial_derivative:
@ static_signal = conditional_expression ;

*/*********************** 4.1.6 allocate_statement **********************/*
allocate_statement:
allocate *allocate_list ;*

allocate_list:
allocate_node
allocate_list , allocate_node

allocate_node:
identifier [constant_expression]

*/*********************** 5.0 Model card definition **********************/*
model_card:
model_header { <model_body> } <;>

model_header:
model *<class_name ::> full_name model_class_name <(<expression_list>)>*

model_body:
model_parameter
model_body model_parameter

model_parameter:
model_assignment ;

model_assignment:
model_lhs model_rhs

model_lhs:
<class_name ::> identifier
model_lhs [constant_expression]
model_lhs . identifier

model_rhs:
= initializer
= model_assignment

*/*********************** #6.0 General-purpose parameter list *************/*
parameter_list:
general_parameter

parameter_list general_parameter

general_parameter:
par_assignment ;
model_attach ;
template_attach ;

par_assignment:
identifier = par_rvalue

model_attach:
<private> **model** *= full_name*

par_rvalue:
expression
par_assignment

*/******************** #8.0 SPICE code *******************************/*
spice_code:
spice *{ sp_lines }*

sp_lines:
sp_line
sp_lines sp_line

sp_line:
** line_of_any_text*
newline
.model *model_name model_class <sp_sets> newline*
+ sp_sets newline

sp_sets:
spice_assign
sp_sets spice_assign

spice_assign:
identifier = spice_constant

spice_constant:
double_constant
- double_constant
+ double_constant

Appendix 4
Alecsis assembler

Virtual processor is one of the main parts of Alecsis simulation engine. It emulates the behaviour of a real hardware processor by executing commands successively according to type. The set of legal instructions for the virtual processor comes with adaptations and changes from the set for MC68020, *Motorola* microprocessor.

Text in AleC++ can be translated into the assembly language (assembler) code if you use option '`-S`' (file with the extension '`.as`') in the program call. Besides, you can write assembler code in the text using command **`asm`** (see chapter 2). It is not likely that the user may need assembler commands for modelling and simulation. However**, it may be necessary to understand assembler if the problems arise when installing Alecsis on different computers**.

When using assembler you can encounter some memory problems, problems with honouring various conventions, etc., thus you need to be careful when using it. The use of assembler is tolerated only for writing of very difficult functions whose time of execution is crucial for the program. Compiler sometimes copies interim results into temporary registers to protect them from deleting. This can be more than necessary sometimes, but compiler uses the safer method. Since Alecsis does not have a multipassage optimizer (besides the **peephole optimizer**, which does not deal with the code as a whole but only with 2-3 neighbouring instructions), there is always room for a shorter and more efficient code written in assembler. Optimizing can, however lead you into dangerous waters, which require the knowledge and understanding of the work of virtual processor. Notice that Alecsis always creates the shortest code from the given text (syntax-directed compilation). Mentioned optimizations refer to rearrangement of some expression in order to eliminate extraneous calculations.

A4.1. Operands in assembler instructions

The cycle of the virtual processor divides into two phases: fetching of operands and execution of instructions. Some conventions regulate what can be an assembler operand. Primary (basic) operands refer to a part of memory called *resource* in Alecsis. That is an internal table containing the base addresses off all regions the potential operands can come from. Operation fetch comes down to two additions: one when indexing resource tables; and the other for addition of the address obtained in that way and the operand offset. The position in the table is set using mnemonics pointing the type of the source, while offset is given by a number. Virtual processor supports the following *resource mnemonics* (mnemonic always goes after the character '`%`').

`%d`*n* - general purpose registers (min. 64 bytes)

`%v`*n* - local memory (allocated using instruction `link`)

`%f`*n* - formal parameter (or action parameter, inside a process)

`%m`*n* - object passed to a function or a process (if it fits the declaration)

`%a`*n* - general purpose address registers (min. 64 bytes)

`%s`*n* - local static memory

`%_`*id* - identifier *id* (name with external linking)

`(%a`*n*`)` - content of memory pointed to by the address register (dereferencing)

`%b`*n* - register pointing to ***resource*** vector

`%e`*n* - memory of the *n*-th element (if it has parameters)

`%n`*n* - memory of the value of the signal at the position *n* (in processes) in the current time instant

The last two operands can appear in processes only, since they refer to object that can be declared only in modules. All elements are indexed by the order of declaration, with the index 0 being the first element. Operand points to the pointer to memory of `action` parameters, if the component has them. The last operand points to the memory containing the value of the signal with the index *n* (formal parameters are indexed as 0, 1, 2,.., while the local ones as -1, -2, -3, ...). To get the real address you need to multiply the index and the length of every signal in bytes.

Beside these operands, all expressions of AleC++ can be found in instructions, including constants and variables. For example:

```
int i, j = 2;
asm movq.l i, j;
```

The previous command copies the content of variable `j` into variable `i`. You should avoid complex expressions, since they can use some registers in the way that creates a conflict with the current instruction.

Operand of instructions for conditional or unconditional branching can be name of a label, which can be inside an **`asm`** region or even outside it, but within the original function or a `process`. The instruction itself can have label using the syntax explained in Chapter 3.

The names of functions in the code are created using the mechanism known as **name mangling** (Chapter 4). You can see how this name is created if you compile the file containing the definition of the function using the option '`-S`', and read its name from the assembler file. For example, function:

```
int foo (int, int, double**, class Point, ...);
```

can be called using instruction:

```
asm jsr %dn, %_foo_iiPPd5Point_e ;
```

where `%d`*n* labels the beginning of memory occupied by the function arguments. The name of the function is prolonged with extensions, that explain which formal parameters are declared for function `foo`. This enables function overloading.

When using temporary registers (`%d`*n*) you can cross over limit of 64 bytes. Actually, all registers with the index larger than `sizeof(double)` will be transferred to the local memory, allocated as much as needed. This especially refers to passing down the arguments during a function call.

A4.2. Assembler instructions

Most of instructions for virtual processor have more than one version. The current implementation of virtual processor supports the following types: **b** (byte), **l** (long), **d** (double). Some instructions do not have all of these types, or have one as default. The type is appended as the extension of the instruction mnemonics (`add.l`, `move.d`). The type gives information about the number of bytes used by the instruction (sometime the meaning of the instruction, too). The exception is the pair **mset**/**movm** where you can copy an unlimited number of bytes).

```
mset  64
movm  %_s1,  %_s2
```

In this example, the first 64 bytes pointed to by the external symbol `s2` are copied to external symbol `s1`.

A4.2.1. Instructions of Alecsis virtual processor

mnemonic	syntax	operation	supported types
add	add.*t* op1, op2	%d0 = op1 + op2	b, l, d
adda	adda.*t* op1, op2	%d0 = op1 += op2	b, l, d
addr	addr.*t* op1, op2	%a0 = op1[op2] Indexing operation has a type, to render multiplication of the index by the type op1 unnecessary	b, l, d
addq	addq.*t* op1, op2	op1 += op2	l
asr	asr.*t* op1, op2	%d0 = op1 << op2	b, l
asra	asra.*t* op1, op2	%d0 = op1 <<= op2	b, l
band	band.*t* op1, op2	%d0 = op1 & op2	b, l
banda	banda.*t* op1, op2	%d0 = op1 &= op2	b, l
bnand	bnand.*t* op1, op2	%d0 = op1 ~& op2	b, l
bnor	bnor.*t* op1, op2	%d0 = op1 ~| op2	b, l
bnot	bnot.*t* op	%d0 = ~op	b, l
bor	bor.*t* op1, op2	%d0 = op1 | op2	b, l
bora	bora.*t* op1, op2	%d0 = op1 |= op2	b, l

bxnor	`bxnor.t op1, op2`	`%d0 = op1 ~^ op2`	b, l
conb	`conb.`*t* `op`	conversion of `op` from type **`byte`** `into` type *t*	b, l, d
cond	`cond.`*t* `op`	conversion of *op* from type **`double`** into type *t*	b, l, d
conl	`conl.`*t* `op`	conversion of *op* from type **`long`** into type *t*	b, l, d
decl	`decl.`*t* `op`	`%d0 = --op`	b, l
decr	`decr.`*t* `op`	`%d0 = op--`	b, l
devb	`dev.`*t* `op1, op2`	`%d0 = op1 / op2`	l, d
deva	`deva.`*t* `op1, op2`	`%d0 = op1 /= op2`	b, l, d
eq	`eq.`*t* `op1, op2`	`%d0 = op1 == op2`	b, l, d
ge	`ge.`*t* `op1, op2`	`%d0 = op1 >= op2`	b, l, d
gt	`gt.`*t* `op1, op2`	`%d0 = op1 > op2`	b, l, d
incl	`incl.`*t* `op`	`%d0 = ++op`	b, l
incr	`incr.`*t* `op`	`%d0 = op++`	b, l
jfn	`jfn op1, findx`	jump to the intrinsic function with the index *`findx`* and arguments beginning from the address `op1`	/
jnz	`jnz label`	jump to label if `%d0 != 0`	/
jp	`jp label`	jump to label	/
jsr	`jsr op1, op2`	jump to function with address `op2` and arguments beginning from address `op1`	/
jz	`jz label`	jump to label if `%d0 == 0`	/
le	`le.`*t* `op1, op2`	`%d0 = op1 <= op2`	b, l, d
lea	`lea op1, op2`	`op1 = &op2`	/
link	`link.`*t* `%b0, size`	shift of the stack for *t*`*size` bytes	b, l, d
lsl	`lsl.`*t* `op1, op2`	`%d0 = op1 << op2`	b, l
lsla	`lsla.`*t* `op1, op2`	`%d0 = op1 <<= op2`	b, l
lt	`lt.`*t* `op1, op2`	`%d0 = op1 < op2`	b, l, d
mod	`mod.`*t* `op1, op2`	`%d0 = op1 % op2`	b, l
moda	`moda.`*t* `op1, op2`	`%d0 = op1 %= op2`	b, l
move	`move.`*t* `op1, op2`	`%d0 = op1 = op2`	b, l, d
movm	`movem op1, op2`	copying of content of `op2` to `op1` - number of copied bytes is determined by the instruction **`mset`**	/
movq	`movq.`*t* `op1, op2`	`op1 = op2`	b, l, d
mset	`mset op`	control of instruction **`movm`**	/
mul	`mul.`*t* `op1, op2`	`%d0 = op1 * op2`	b, l, d
mula	`mula.`*t* `op1, op2`	`%d0 = op1 *= op2`	b, l, d
neg	`neg.`*t* `op`	`%d0 = - op`	b, l, d
neq	`neq.`*t* `op1, op2`	`%d0 = op1 != op2`	b, l, d
not	`not.`*t* `op`	`%d0 = !op`	b, l, d
rts	`rts`	exit from a procedure	/
sub	`sub.`*t* `op1, op2`	`%d0 = op1 - op2`	b, l, d
suba	`suba.`*t* `op1, op2`	`%d0 = op1 -= op2`	b, l, d
subq	`subq.l op1, op2`	`op1 -= op2`	l
unlk	`unlk %b0`	return of the local memory stack during the exit from a procedure	/
xor	`xor.t op1, op2`	`%d0 = op1 ^ op2`	b, l

xora	xora.t op1, op2	%d0 = op1 ^= op2	b, l

A4.2.2. Instructions of Alecsis virtual coprocessor

In the previous section, basic instruction set of virtual processor is given. Beside those instruction, virtual processor has something you can call "coprocessor". Those are additional instructions supporting some of most frequently used functions. This makes a program more effective, especially in the case of mathematical functions, which are used often in modelling of analogue circuits.

mnemonic	syntax	operation	supported types
putchar	putchar.l op	%d0=putchar (op)	l
fputc	fputc.l op1, op2	%d0 = fputc (op1, op2)	l
getchar	getchar.l	%d0 = getchar()	l
fgetc	fgetc.l op	%d0 = fgetc (op)	l
strcpy	strcpy.l op1, op2	%d0 = strcpy(op1, op2)	l
strcmp	strcmp.l op1, op2	%d0 = strcmp(op1, op2)	l
strlen	strlen.l op	%d0 = strlen(op)	l
malloc	malloc.l op	%d0 = malloc (op)	l
calloc	calloc.l op1, op2	%d0 = calloc (op1, op2)	l
free	free op	free (op)	/
attr	attr.l indx, offset	returns the address of user-defined attributes for signals with the position indx and offset offset	l
slen	slen.l indx, offset	returns the length of the signal-vector with the position indx and offset offset	l
fabs	fabs.d op	%d0 = fabs (op)	l, d
exp	exp.d op	%d0 = exp (op)	d
log	log.d op	%d0 = log (op)	d
log10	log10.d op	%d0 = log10 (op)	d
pow	pow.d op	%d0 = pow (op)	d
sqrt	sqrt.d op	%d0 = sqrt (op)	d
sin	sin.d op	%d0 = sin (op)	d
cos	cos.d op	%d0 = cos (op)	d
tan	tan.d op	%d0 = tan (op)	d
asin	asin.d op	%d0 = asin (op)	d
acos	acos.d op	%d0 = acos (op)	d
atan	atan.d op	%d0 = atan (op)	d
atan2	atan2.d op1, op2	%d0 = atan2 (op1, op2)	d
sinh	sinh.d op	%d0 = sinh (op)	d
cosh	cosh.d op	%d0 = cosh (op)	d
tanh	tanh.d op	%d0 = tanh (op)	d
floor	floor.d op	%d0 = floor (op)	d
ceil	ceil.d op	%d0 = ceil (op)	d

You can use coprocessor instructions by honouring standard conventions.

A4.3. Conventions on passing parameters to functions

The mechanism of function call and the return from functions, used by the virtual processor, will be explained on the following example.

```
int z;

main () {
   int x, y;
   z = test (x, y);
}

test (int i, int j) {
   int k;
   k = i + j;
   return (k);
}
```

This code would be compiled as follows (comments in the code are added):

```
_main:
   link.l %b0, 4      // allocating 16 bytes of local space
   movq.l %d8, %v0    // placing variable i (%v0) into the first
       // available register after the accumulator (%d0 to %d7)
   movq.l %d12, %v4   // placing variable j into the next one
   jsr  %d8, _test_ii //call of func. test (the name is completed)
   movq.l %_z, %d0    // result returned via %d0
L0:
   unlk  %b0          // freeing local space
   rts                // end of function

_test_ii:
   link.b %b0, 4     // allocating 4 bytes of local space
   add.l  %f0, %f4   // adding of formal variables i and j
   movq.l %v0, %d0   // storing the result into the variable k
   movq.l %d0, %v0   // return of the result
   jp      L0        // compulsory jump to the output label (to
                     // free the space allocated using link
L0:
   unlk %b0          // freeing local space
   rts               // end of function
```

The arguments are passed in the following manner - they are lined up continually into the register %d*n*, starting from the first free position (the lowest position is `%d8`, since the accumulator occupies the first 8 bytes). Instruction `link` allocates space for all local variables and all interim results that were on locations `%d8`+*n*. In our example function `main` has two local variables of type `int` (2x4 bytes) and uses two positions of a register `%d` to pass arguments (2x4 bytes, totalling 16 - since we used long variant of instruction `link`, this number is divided by `sizeof(long)`). Function `test` allocates space only for its local variable `k`. The result of the function is returned through the accumulator (from `%d0` to `%d7`). After that, the program jumps to label `L0` where it frees local space, and exits from the function. The results larger than 8 bytes return to address `%f0` (address of formal parameters), using instruction `movm`.

The previous example can be realized using combined AleC++/assembler syntax:

```
int z;
```

```
main () {
    int x, y;
    asm {
        movq.l %d8, x
        movq.l %d12, y
        jsr      %d8, %_test_ii
        movq.l z, %d0
    }
}

test (int i, int j) {
    int k;
    asm {
        add.l i, j
        movq.l k, %d0
    }
    return (k);
}
```

This example leaves instructions `link` and `inlk` to the compiler (this is a standard procedure when using **`asm`** command). This applies in both cases to command `return`, too

In pointer arithmetic, you should be careful when dealing with address registers. Register `%a0` is reserved for vector indexing. Therefore, the code:

```
int i,j, a[10], b;
b = a[i+j];
```

compiles to

```
add.l i, j
addr.l a, %d0
movq.l b, (%a0)
```

which means that the instruction `addr` puts the address `&a + sizeof(long)*(i+j)` into the address register `%a0`. The following instruction copies the **content** of that address in the register into variable `b`.

Parentheses can dereference only address registers. To dereference a pointer, you need to transfer it into an address register:

```
int *i, j[10];
j[2] = *i;
```

This code is equivalent to the following code:

```
addr.l j, 2
movq.l %a4, i
movq.l (%a0), (%a4)
```

Note that pointer occupy 4 bytes, so the first free place was `%a4`, after `%a0` had been used.

Instruction `lea` does the **referencing**:

```
int i, j;
j = &i;

lea %d0, i
```

```
movq.l j, %d0
```

Compiler implements referencing in two steps, but it is the standard procedure for the optimizer to merge two steps into one: `lea j, i`.

Built-in functions are called using command `jfn`, and they cannot be mixed with ordinary functions, since their address cannot be obtained. The first operand is the address of the first argument, while the second one is an integer constant used for indexing. Indexes of all intrinsic functions, that are not instructions, are in the file `asm.h` in directory `alecsis/include`. The following is the content of the file.

```
#define _printf        0
#define _fprintf       1
#define _sprintf       2
#define _fflush        3
#define _fopen         4
#define _fclose        5
#define _feof          6
#define _fseek         7
#define _ftell         8
#define _fread         9
#define _fwrite        10
#define _rewind        11
#define _exit          12
#define _system        13
#define _warning       16
#define _drand         17
#define _get_info      21
#define _atof          27
#define _atoi          28
```

The missing indices are used for internal system functions, which only compiler can call. By appending this library you can write:

```
movq.l %d8, "Hello, world!\n"
jfn    %d8, _printf
```

which has the same effect as AleC++ command:

```
printf("Hello, world!\n").
```

You can find all other details linked with using assembler by compiling the source code using option '`-S`' and by direct comparison of source and compiled code. Notice that you cannot use directly the code obtained in this manner. Assembler instructions need to be inside **asm** command for compiler to accept them. The closing example will be a recursive function for calculating the factorial of 170 in double precision:

```
#include <alec.h>

double factor (double i){
    if (i<=1) return 1.0;
    return i * factor (i-1.);
}

int main() {
    double i=170.;
    printf("\tfactor(%g)=%g\n", i, factor (i));
}

//
```

```
// Alecsis assembler code
//
// optimization off
//

// function factor_d

_factor_d:
    link.b   %b0, 8
    le.d     %f0, 1
    movq.l   %d0, %d0
    jz  L1
    movq.d   %d0, 1
    jp  L0
L1:
    sub.d    %f0, 1
    movq.d   %v0, %d0
    jsr      %v0, _factor_d
    mul.d    %f0, %d0
    movq.d   %d0, %d0
    jp  L0
L0:
    unlk     %b0
    rts

//function main

_main:
    link.b   %b0, 28
    movq.d   %v0, 170
    movq.l   %v8, "\tfactor(%g)=%g\n"
    movq.d   %v12, %v0
    movq.d   %v20, %v0
    jsr           %v20, _factor_d
    movq.d   %v20, %d0
    jfn           %v8, 0
L0:
    unlk     %b0
    rts
```

Appendix 5
Model card parameters for built-in components

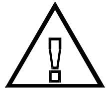
In this Appendix, names and default values of model card parameters are given for built-in analogue components. These are SPICE models of diode, MOSFET, BJT, and JFET. In SPICE manuals more detailed explanations of these models and model card parameters can be found. Nevertheless, there are different versions of SPICE, and we hope this list of model card parameters can be useful to determine which version of SPICE model is implemented in Alecsis.

If you need some version of the model that is not built into Alecsis, you have to define new model in AleC++.

Note: In the parameter tables, some of the parameters are dummy, i.e. they have no meaning. They are given here for completeness, as memory is allocated for them (as in SPICE). Some of them are used for results of parameters preprocessing.

A5.1. Diode model card parameters

SPICE 2G6 diode model is built into Alecsis.



Physical units are not given in the following table. We will give these units in new versions of this Manual.

Table A5.1. Diode model card parameters in Alecsis.

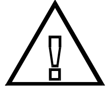
name	default	unit	meaning
		-	dummy, not used
is	1e-14		saturation current
rs	0.0		parasitic resistance
n	1.0		emission coefficient
tt	0.0		transit time
cjo	0.0		zero-bias p-n capacitance
vj	1.0		p-n potential
m	0.5		p-n grading coefficient
eg	1.11		bandgap voltage
xti	3.0		IS temperature coefficient
kf	0.0		flicker noise coefficient
af	1.0		flicker noise exponent
fc	0.5		forward-bias depletion capacitance coefficient
bv	0.0		reverse breakdown "knee" voltage
ibv	1e-3		reverse breakdown "knee" current
Following parameters are read from the model card, but are not used in the current version of the model:			
isr	-		recombination current parameter
nr	-		emission coefficient for ISR
ikf	-		high-injection "knee" current
nbv	-		reverse breakdown ideality factor
ibvl	-		low-level reverse breakdown "knee" current
nbvl	-		low-level reverse breakdown ideality factor
tikf	-		IKF temperature coefficient (linear)
tbv1	-		BV temperature coefficient (linear)
tbv2	-		BV temperature coefficient (quadratic)
trs1	-		RS temperature coefficient (linear)
trs2	-		RS temperature coefficient (quadratic)

A5.2. MOSFET model card parameters

Alecsis has four versions (levels) of MOS models. These are level 1, level 2, level 3 and level 13 models. The first three are standard SPICE models. Fourth model is BSIM model (**B**erkeley **S**hort-Channel **I**GFET **M**odel), which is denoted as level 13 in HSPICE.

A5.2.1. MOSFET level 1, 2 and 3 parameters

SPICE 2G6 MOSFET level 1, 2, and 3 models are built into Alecsis.



Parameter explanations are not given in the following table. We will give these explanations in new versions of this Manual.

Table A5.2. MOSFET level 1, 2, and 3 model card parameters in Alecsis.

name	default	unit	meaning
level	1	-	
gamma	-	$V^{1/2}$	
nss	-	$1/cm^2$	
nsub	1.0e15	$1/cm^3$	
phi	-	V	
tpg	1.0	-	
vto	-	V	
af	1.0	-	
kf	-	-	
rd	-	Ω	
rs	-	Ω	
rsh	-	Ω/square	
cgso	-	F/m	
cgdo	-	F/m	
cgbo	-	F/m	
tox	1.0e-7	m	
cj	-	F/m^2	
cjsw	-	F/m	
mj	0.5	-	
mjsw	0.33	-	
pb	0.8	V	
fc	0.5	-	
ld	-	m	
kp	-	A/V^2	
lambda	-	1/V	
delta	-	-	
neff	1.0	-	
nfs	-	$1/cm^2$	
ucrit	1.0e4	V/cm	
uexp	-	-	
uo	600.0	$cm^2/(Vs)$	
vmax	-	m/s	
xj	-	m	
is	1.0e-14	A	
js	-	A/m^2	
kappa	0.2	-	
theta	-	1/V	
cox	-	F/m^2	
eta	-	-	
vbi	-	V	
xqc	-	-	
xd	-	-	For use of macromodels.
fnarrow	-	-	
vt	-	-	
xd2	-	-	

Following parameters are read from the model card, but are not used in the current version of the model:			
rg	-	Ω	
rb	-	Ω	
rds	-	Ω	
jssw	-	A/m	
n	-	–	
pbsw	-	V	
cbd	-	F	
cbs	-	F	
tt	-	s	
wd	-	m	
utra	-	-	
The following two parameters are used, if they are not given when MOS transistor is invoked (connected):			
l	0.0	m	channel length
w	0.0	m	channel width

A5.2.2. BSIM parameters (level 13)

HSPICE MOSFET level 13 model is built into Alecsis.

Table A5.3. MOSFET level 13 model card parameters in Alecsis.

name	default	unit	meaning
level		-	mosfet model level selector, 13 for HSPICE BSIM
vfb0	-1.0641	V	flatband voltage and its length and width sensitivities
lvfb	1.71979e-1	V μm	
wvfb	1.11454e-1	V μm	
phi0	7.95392e-1	V	two times the Fermi potential, its length and width sensitivities
lphi	0.0	V μm	
wphi	0.0	V μm	
k1	1.10425	$V^{1/2}$	root-vbs threshold coefficient, its length and width sensitivities
lk1	-4.3371e-1	$V^{1/2}$ μm	
wk1	-9.8518e-2	$V^{1/2}$ μm	
k2	1.93126e-1	-	linear vbs threshold coefficient, its length and width sensitivities
lk2	4.14269e-4	μm	
wk2	-6.0274e-2	μm	
eta0	-4.7124e-3	-	linear vds threshold coefficient, its length and width sensitivities
leta	-1.0565e-2	μm	
weta	1.08645e-2	μm	
muz	6.00853e2	cm^2/(Vs)	low drain field first order mobility
dl0	6.2438e-1	μm	difference between drawn poly and electrical
dw0	1.0384	μm	difference between drawn diffusion and electrical
u00	5.11222e-2	1/V	gate field mobility reduction factor, its length and width sensitivities
lu0	1.73108e-1	(1/V) μm	
wu0	-5.9804e-2	(1/V) μm	
u1	-2.3954e-1	μm/V	drain field mobility reduction factor, its length and width sensitivities
lu1	2.91101	(μm/V) μm	
wu1	-5.3638e-2	(μm/V) μm	
x2m	4.66158	$(cm/V)^2$ / s	vbs correction to low field first order mobility, its length and width sensitivities
lx2m	-8.0305	($(cm/V)^2$ / s) μm	
wx2m	5.54267	($(cm/V)^2$ / s) μm	

x2e	-9.142e-4	1/V	vbs correction to linear vds threshold coefficient, its length and width sensitivities
lx2e	1.23113e-2	(1/V) µm	
wx2e	2.4326e-3	(1/V) µm	
x3e	1.05704e-4	1/V	vds correction to linear vds threshold coefficient, its length and width sensitivities
lx3e	1.04115e-2	(1/V) µm	
wx3e	-2.5834e-3	(1/V) µm	
x2u0	2.68363e-4	$1/V^2$	vbs reduction to gate field mobil. reduction factor, its length and width sensitivities
lx2u0	-1.5668e-3	$(1/V^2)$ µm	
wx2u0	-8.5052e-4	$(1/V^2)$ µm	
x2u1	-7.2567e-2	$\mu m/V^2$	vbs reduction to drain field mobil. reduction factor, its length and width sensitivities
lx2u1	1.10182e-1	$(\mu m/V^2)$ µm	
wx2u1	5.66859e-2	$(\mu m/V^2)$ µm	
mus	5.49834e2	$cm^2/(Vs)$	high drain field mobility, its length and width sensitivities
lms	1.77273e3	$cm^2/(Vs)$ µm	
wms	-9.0196e1	$cm^2/(Vs)$ µm	
x2ms	-1.6724e1	$(cm/V)^2 / s$	vbs reduction to high drain field mobility, its length and width sensitivities
lx2ms	8.98504	$((cm/V)^2 / s)$ µm	
wx2ms	2.8234e1	$((cm/V)^2 / s)$ µm	
x3ms	4.86164	$(cm/V)^2 / s$	vds reduction to high drain field mobility, its length and width sensitivities
lx3ms	1.56629e1	$((cm/V)^2 / s)$ µm	
wx3ms	-6.57	$((cm/V)^2 / s)$ µm	
x3u1	7.76925e-3	$\mu m/V^2$	vds reduction to drain field mobility reduction factor, its length and width sensitivities
lx3u1	-1.094e-1	$(\mu m/V^2)$ µm	
wx3u1	-8.3353e-3	$(\mu m/V^2)$ µm	
toxm	2.5e-2	µm	gate oxide thickness
tempm	25.0	ºC	reference temperature of model
vddm	5.0	V	critical voltage for high drain field mobility reduction
cgdom	1.5e-9	F/m	gate to drain parasitic capacitance; f/m of width
cgsom	1.5e-9	F/m	gate to source parasitic capacitance; f/m of width
cgbom	2.0e-10	F/m	gate to bulk parasitic capacitance; f/m of length
xpart	1.0	-	selector for gate capacitance charge sharing coefficient
dum1	0.0	-	dummy, not used
dum2	0.0	-	dummy, not used
n0	1.5	-	low field weak inversion gate drive coefficient, -||-, value of 200 for n0 disables weak inversion calculation
ln0	0.0	µm	
wn0	0.0	µm	
nb0	0.1	1/V	vbs reduction to low field weak inversion gate drive coefficient., its length and width sensitivities
lnb	0.0	(1/V) µm	
wnb	0.0	(1/V) µm	
nd0	0.0	1/V	vds reduction to low field weak inversion gate drive coefficient., its length and width sensitivities
lnd	0.0	(1/V) µm	
wnd	0.0	(1/V) µm	
rshm	50.0	Ω/square	sheet resistance / square
cjm	4.5e-5	F/m^2	zero-bias bulk junction bottom capacitance
cjw	0.0	F/m	zero-bias bulk junction sidewall capacitance
ijs	1.0e-4	A/m^2	bulk junction saturation current
pj	0.8	V	bulk junction bottom potential
pjw	0.6	V	bulk junction sidewall potential
mj0	0.5	-	bulk junction bottom grading coefficient
mjw	0.33	-	bulk junction sidewall grading coefficient
wdf	2.0e-6	m	default width of the layer
ds	0.5	m	average variation of size due to side etching or mask compensation

A5.3. BJT model card parameters

SPICE 2G6 bipolar junction transistor (BJT) model is built into Alecsis.

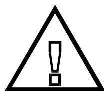
Table A5.4. BJT model card parameters in Alecsis.

name	default	unit	meaning
		-	dummy, not used
is	1.0e-16	A	saturation current
bf	100.	-	ideal maximum forward current gain
nf	1.	-	forward current emission coefficient
vaf	0. (means ∞)	V	forward early voltage
ikf	0. (means ∞)	A	corner for forward beta high-current roll-off
ise	0.	A	base-emitter leakage saturation current
ne	1.5	-	base-emitter leakage emission coefficient
br	1.	-	ideal maximum reverse current gain
nr	1.	-	reverse current emission coefficient
var	0. (means ∞)	V	reverse early voltage
ikr	0. (means ∞)	A	corner for reverse beta high-current roll-off
isc	0.	A	base-collector leakage saturation current
nc	2.	-	base-collector leakage emission coefficient
		-	dummy, not used
		-	dummy, not used
rb	0.	Ω	zero bias base resistance
irb	0. (means ∞)	A	current where base resistance falls halfway to its minimum value
rbm	0. (means rb)	Ω	minimum base resistance at high currents
re	0.	Ω	emitter resistance
rc	0.	Ω	collector resistance
cje	0.	F	zero-bias base-emitter depletion capacitance
vje	0.75	V	base-emitter built-in potential
mje	0.33	-	base-emitter junction grading coefficient
tf	0.	s	ideal forward transit time
xtf	0.	-	coefficient for bias dependence of tf
vtf	0. (means ∞)	V	voltage describing vbc depencence of tf
itf	0.	A	high-current parameter for effect on tf
ptf	0.	º	excess phase at f=1/(2*π*tf)
cjc	0.	F	zero-bias base-collector depletion capacitance
vjc	0.75	V	base-collector built-in potential
mjc	0.33	-	base-collector junction grading coefficient
xcjc	1.	-	fraction of base-collector depletion capacitance connected to internal base node
tr	0.	s	ideal reverse transit time
		-	dummy, not used
		-	dummy, not used
		-	dummy, not used
		-	dummy, not used
cjs	0.	F	zero-bias collector-substrate capacitance
vjs	0.75	V	substrate-junction built-in potential
mjs	0.5	-	substrate-junction exponential factor
xtb	0	-	forward and reverse beta temperature coefficient
eg	1.11	eV	energy gap for temperature effect on is
xti	3.	-	saturation current temperature exponent
kf	0.	-	flicker noise coefficient

af	1.	-	flicker noise exponent
fc	0.5	-	coefficient for forward-bias depletion
Following parameters are read from the model card, but are not used in the current version of the model:			
nk		-	high-current roll-off coefficient
iss		A	substrate p-n saturation current
ns		-	substrate p-n emission coefficient
qco		C	epitaxial region charge factor
rco		Ω	epitaxial region resistance
vo		V	carrier mobility "knee" voltage
gamma		-	epitaxial region doping factor
tre1		1/ºC	RE temperature coefficient(linear)
tre2		$1/(ºC)^2$	RE temperature coefficient(quadratic)
tbr1		1/ºC	RB temperature coefficient(linear)
tbr2		$1/(ºC)^2$	RB temperature coefficient(quadratic)
trm1		1/ºC	RBM temperature coefficient(linear)
trm2		$1/(ºC)^2$	RBM temperature coefficient(quadratic)
trc1		1/ºC	RC temperature coefficient(linear)
trc2		$1/(ºC)^2$	RC temperature coefficient(quadratic)

A5.4. JFET model card parameters

SPICE JFET model is built into Alecsis.



Physical units and parameter explanations are not given in the following table for most of the parameters. We will give these units and explanations in new versions of this Manual.

Table A5.5. JFET model card parameters in Alecsis.

name	default	unit	meaning
		-	dummy, not used
vto	-2.0		
beta	1e-4		
lambda	0.0		
rd	0.0		
rs	0.0		
cgs	0.0		
cgd	0.0		
pb	1.0		
is	1e-14		
kf	0.0		
af	1.0		
fc	0.5		
		-	dummy, not used
		-	dummy, not used
		-	dummy, not used

		-	dummy, not used
Following parameters are read from the model card, but are not used in the current version of the model:			
		-	dummy, not used
		-	dummy, not used
		-	dummy, not used
`n`			gate p-n emission coefficient
`isr`			gate p-n recombination current parameter
`nr`			emission coefficient for ISR
`alpha`			ionization coefficient
`vk`			ionization "knee" voltage
`m`			gate p-n grading coefficient
`vtotc`			VTO temperature coefficient
`betatce`			BETA exponential temperature coefficient
`xti`			IS temperature coefficient

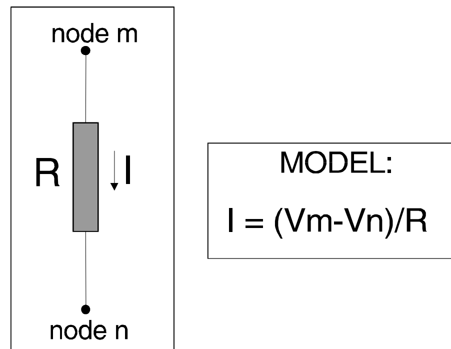
Appendix 6
Analogue simulation examples

Since most of the users prefer learning from examples, in this Appendix some simple electronic analogue and mechanical examples are given. Usage of model cards, `clone` command, and other advanced modelling techniques are not described here.

A6.1. Electronic models

Some of the models described here are already built-in components of Alecsis, so you do not need to describe such models. They are here given as introduction, as it is always easier to start from simple examples.

A6.1.1. Resistor



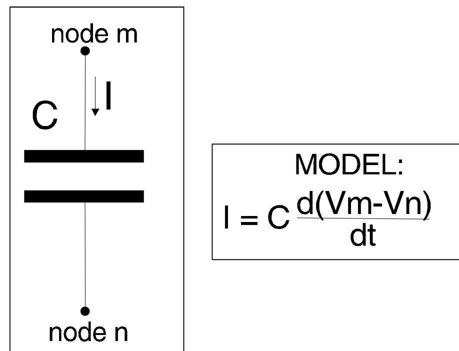
Model stamp:

	Vm	Vn	rhs
m	1/R	-1/R	
n	-1/R	1/R	

Model code:

```
module MyResistor (node m, n)
{
   action (double value=0.0) {
      // avoid division by zero
      process initial {
         if (!value)
            warning ("zero resistance", 1);
      }

      process initial {
         double G;
         // conductance for equation matrix
         G = 1./value;
         // equation (stamp):
         eqn {m,n}.i = G*{m,n}.v;
      }
   }
}
```

A6.1.2. Capacitor



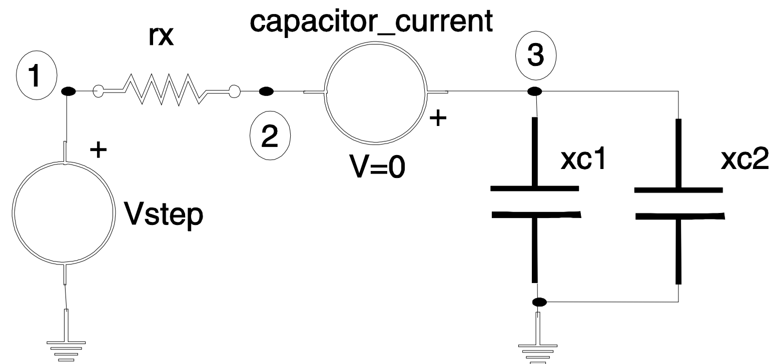
Model stamp:

	Vm	Vn	rhs
m	C/h	-C/h	(C/h)*(Vmp-Vnp)
n	-C/h	C/h	(C/h)*(Vmp-Vnp)

Model code:

```
module MyCapacitance (node m, n) {
   // capacitance C=0.0pF -> default value
   action (double C=0.0) {
      process per_moment {
         //equation (stamp) matrix!
         eqn {m,n}.i = C*ddt{m,n}.v;
      }
   }
}
```

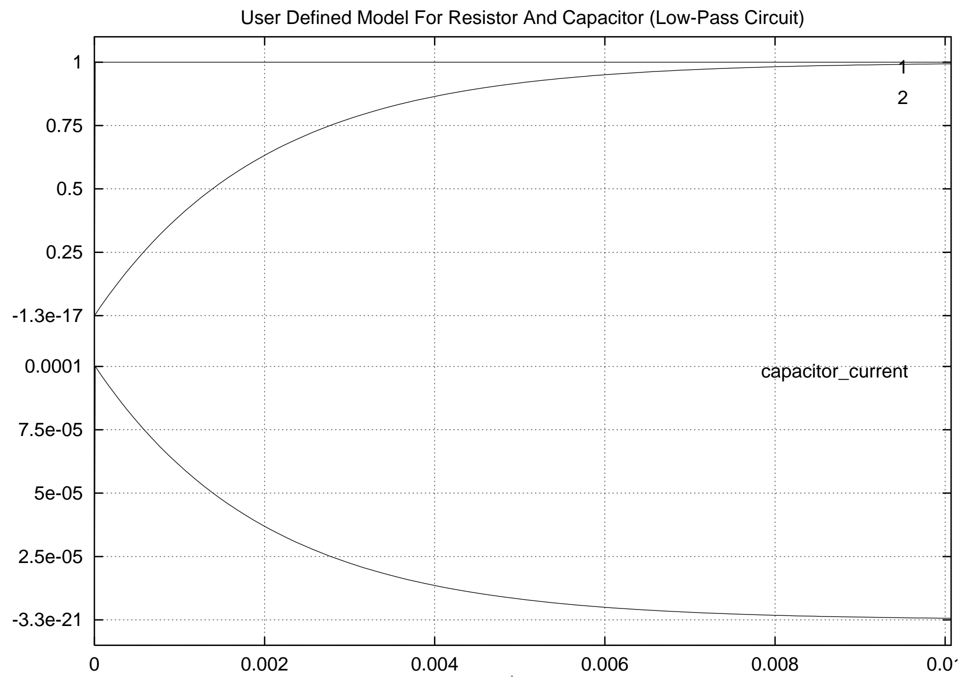
A6.1.3. Example 1



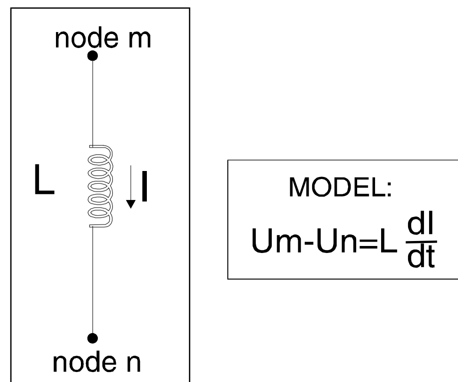
System description:

```
#include <alec.h>
#define Period 10 ms
root module eq ()
{
   // declarations
   MyResistor r;
   MyCapacitance xc1, xc2;
   vgen capacitor_current; // built-in voltage source
   vpwl vin;               // built-in piecewise linear volt. gen.

   // connections
   rx(1,2) 10k;
   capacitor_current (2,3) 0V; // used to measure current
   xc1(3,0) 100nF;
   xc2(3,0) 100nF;
   vin (1, 0) { 0,0; 1ns, 1; Period, 1; }

   // simulation control
   timing { tstop = Period;   a_step=Period/1000; }
   plot { caption
        "User Defined Model For Resistor
         And Capacitor (Low-Pass Circuit)";
         node 1, 2;    // these two nodes use the same scaling
         current capacitor_current; }
}
```

Simulation results:


User Defined Model For Resistor And Capacitor (Low-Pass Circuit)
1
0.75
0.5
0.25
-1.3e-17
0.0001
7.5e-05
5e-05
2.5e-05
-3.3e-21
0
0.002
0.004
0.006
0.008
1
2
capacitor_current

A6.1.4. Inductor



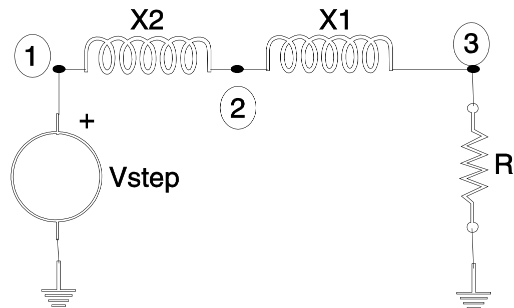
Model stamp. Current must be independent variable in the system of equations:

	Vm	Vn	I	rhs
m			+1	
n			-1	
new	1	-1	L/h	(L/h)*Ip

Model description. Branch current is returned using name of the module:

```
module current MyInductance (node m, n) {
   // inductance L=0.0uH -> default value
   action (double L=0.0) {
      // zero inductance check
      process initial {
         if (!L)
             warning ("zero inductance", 1);
      }

      process per_moment {
         // equation (stamp)
         eqn MyInductance, {m.n}.v = L*ddt{MyInductance};
      }
   }
}
```

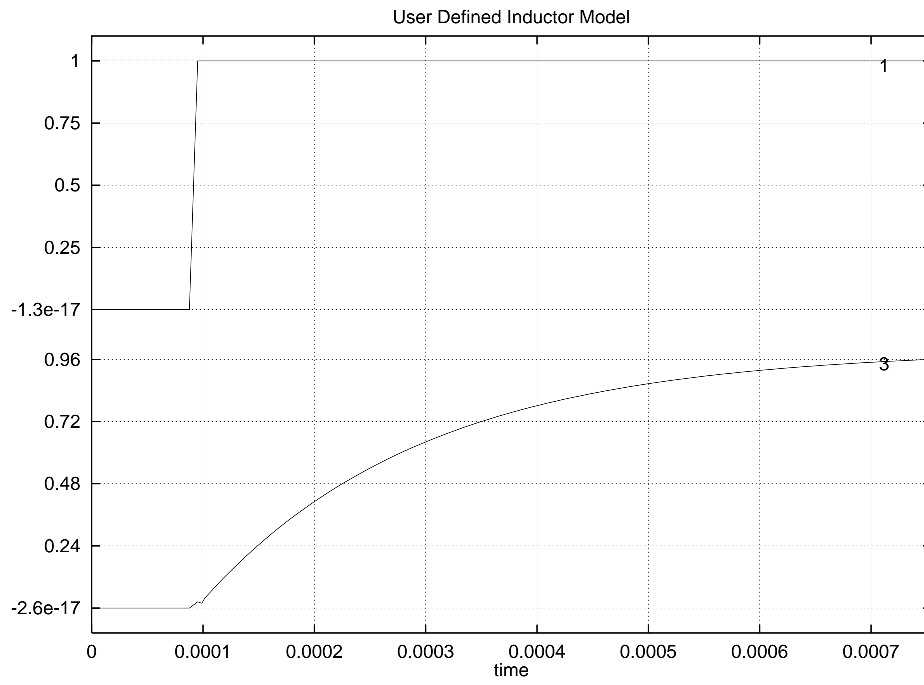
A6.1.5. Example 2



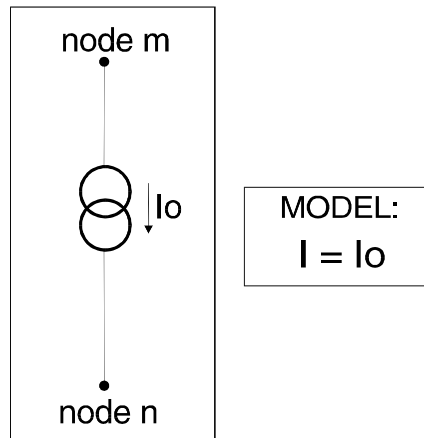
Circuit description:

```
#include <alec.h>
#define Period 750 us

root eq ()
{
   MyResistor r;
   MyInductance x1,x2;
   vpwl vin;      // built-in

   r(3,0) 1.;
   x1(2,3) L=100uH;
   x2(1,2) L=100uH;

   vin (1, 0) { 0,0; Period/8.,0;
                Period/8.,1;  Period, 1; }
   timing { tstop = Period;
            a_step=Period/1000; }
   plot { caption
            "User Defined Inductor Model";
         node 1; node 3;     // two separate waveforms
       }
}
```

Simulation results:


User Defined Inductor Model
1
0.75
0.5
0.25
-1.3e-17
0.96
0.72
0.48
0.24
-2.6e-17
0
0.0001
0.0002
0.0003
0.0004
0.0005
0.0006
0.0007
time
1
3

A6.1.6. Current source



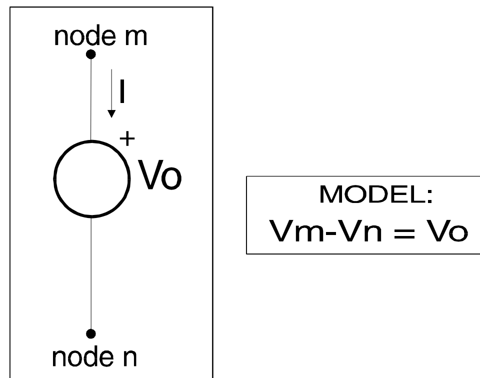
Model stamp:

node	Vm	Vn	rhs
m			-Io
n			+Io

Model description:

```
module Current_source (node m, n) {
   // default current Io=0.0
   action double Io=0.0) {
      virtual process initial {
         eqn {m,n}.i = Io;
      }
   }
}
```

A6.1.7. Voltage source



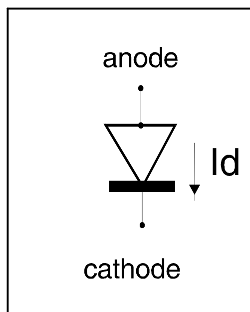
Model stamp. Branch current has to be independent quantity in the system of equations:

node	Vm	Vn	I	rhs
m			+1	
n			-1	
I	1	-1		Vo

Model description:

```
module current Voltage_source (node m, n){
   // default voltage Vo=0.0
   action (double Vo = 0.0) {
      virtual process initial {
         eqn Voltage_source, {m,n}.v = Vo;
      }
   }
}
```

A6.1.8. Diode

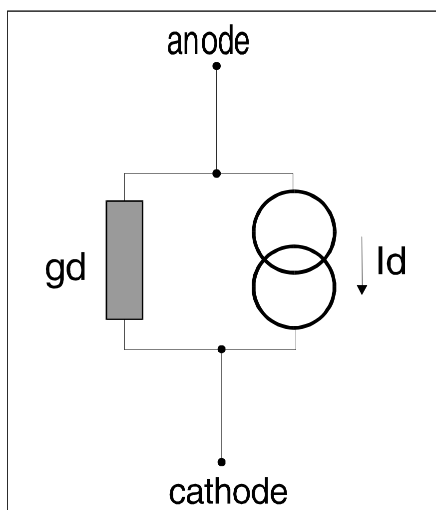


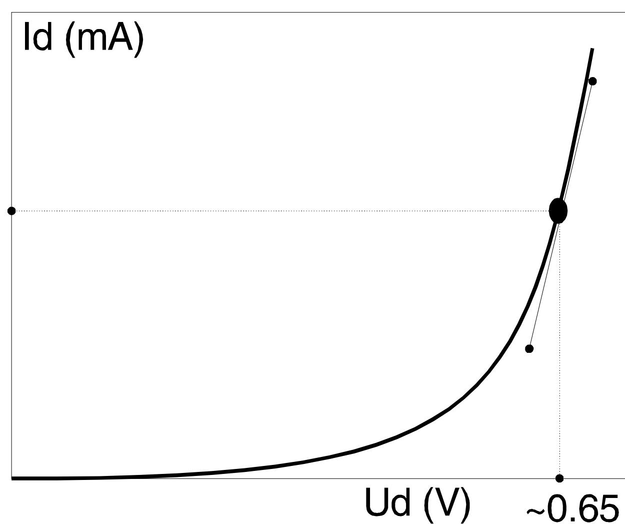
MODEL:

$$gd = \frac{d(Id)}{d(Vd)}$$

$$Vd = Vanode - Vcathode$$

$$Id - Idn = gd^{*}(Vd - Vdn)$$





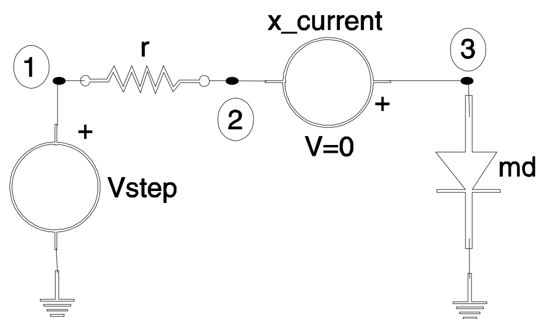
Model with user defined linearized structure. Real diode model should have more parameters and model card, this is simplified description:

```
module MyDiode1 (node m, n) {
   action (double Is=1e-14) {
      process per_iteration {
         double gd,Vd_p,Id_p;
         double Vt=25.8mV;

         // voltage from the last iteration
         Vd_p = m-n;
         // current from the last iteration
         Id_p = Is*exp((m-n)/Vt);
         // slope of the linearized equation - d(Id)/d(Vd)
         gd = Id_p/Vt;

         // linearized equation definition
         eqn {m,n}.i = gd*{m,n}.v - gd*Vd_p + Id_p;

         /* alternative description of the same equation:
            eqn m:  gd*{m}-gd*{n} = -Id_p+gd*Vd_p;
            eqn n: -gd*{m}+gd*{n} = +Id_p+gd*Vd_p;
         */
      }
   }
}
```

Automated description of the linearized model using `nlcgen` - nonlinear current generator:

```
module MyDiode2 (node m, n){
   nlcgen Id;
   Id (m,n,m,n);//connected between m and n, controlled by m and n

   action (double Is=1e-14) {
      process per_iteration {
         double id, gd;
         double Vt=25.8mV;

         id = Is*(exp((m-n)/Vt)-1);
         gd = (id+Is)/Vt;

         // linearized equation with partial derivatives
         nlcgen Id = id { @m=gd; @n=-gd; }

         /* alternative description - automated linearization,
            gd is not necessary:
            nlcgen Id=id;
         */
      }
   }
}
```

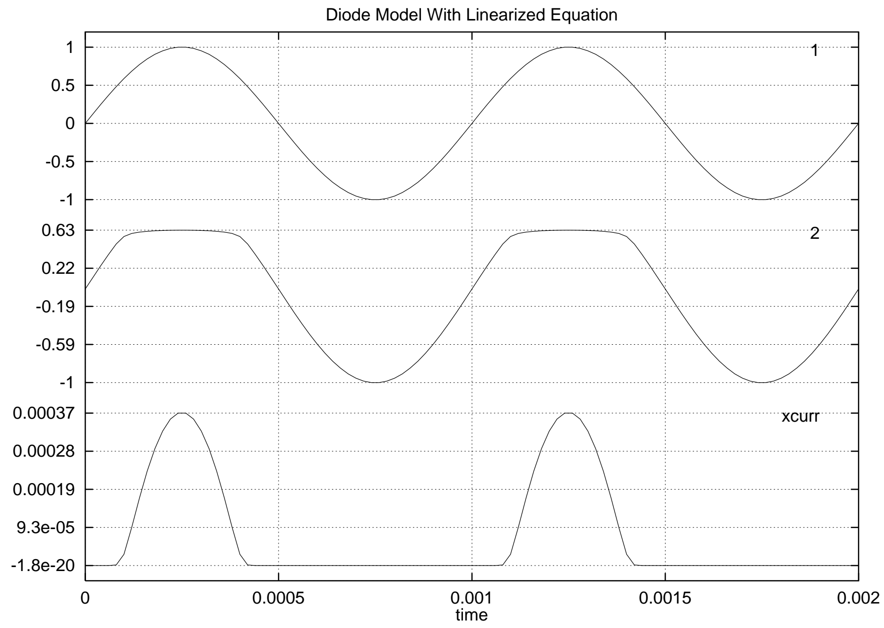
A6.1.9. Example 3



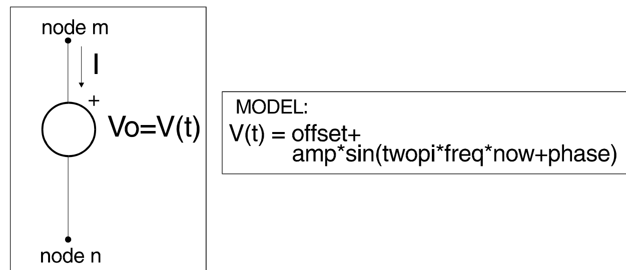
Circuit description:

```
#include <alec.h>
#define Period 2 ms

root module eq ()
{
   MyResistor r;
   MyDiode1 md;
   vsin vin;
   Voltage_source xcurr;

   md(3,0);
   xcurr (2,3) 0V;
   r(2,1) 1e3;
   vin (1, 0) { amp=1V; freq=1kHz; }

   timing { tstop = Period;           a_step=Period/100; }
   plot { caption "Diode Model With Non-Linear Current Generator";
          node 1; node 2;
          current xcurr;
       }
}
```

Simulation results:


Diode Model With Linearized Equation
1
0.5
0
-0.5
-1
0.63
0.22
-0.19
-0.59
-1
0.00037
0.00028
0.00019
9.3e-05
-1.8e-20
1
2
xcurr
0
0.0005
0.001
0.0015
0.002
time

A6.1.10. Sinusoidal voltage generator



Model description:

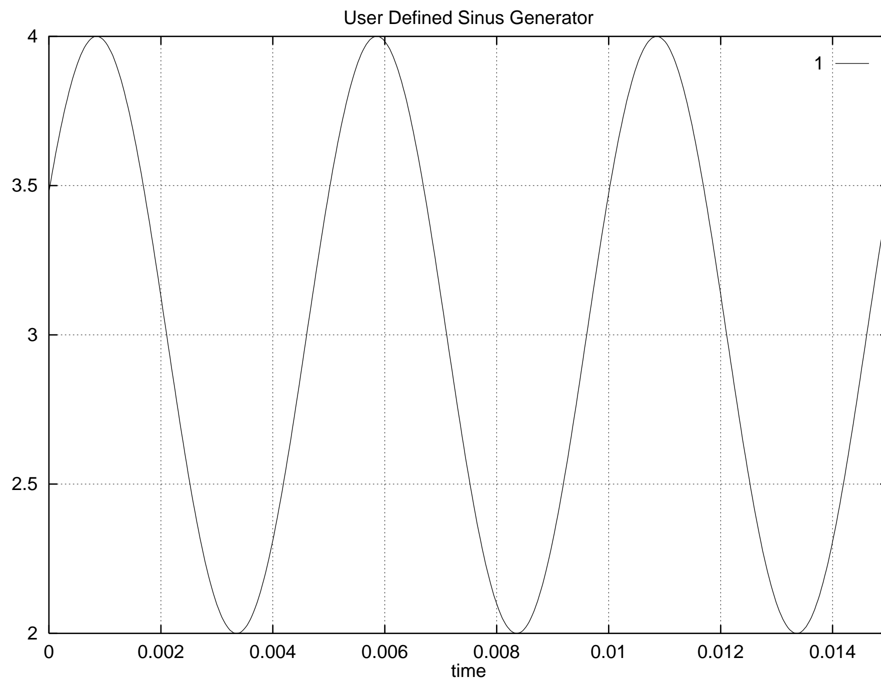
```
#define twopi 6.282

module current Sinus(node m, n) {
   vgen Sinus; // built-in const. voltage src. is used as a basis

   return Sinus(m,n); // connection - branch current of vgen
                      // is returned using name of the module

   // default parameter definition
   action per_moment (double amp=1.0v,
                      double freq=1kHz,
                       double phase=0.0rad,
                       double offset = 0V){
      Sinus->value = offset + amp*sin(twopi*freq*now+phase);
   }
}
```

A6.1.11. Example 4

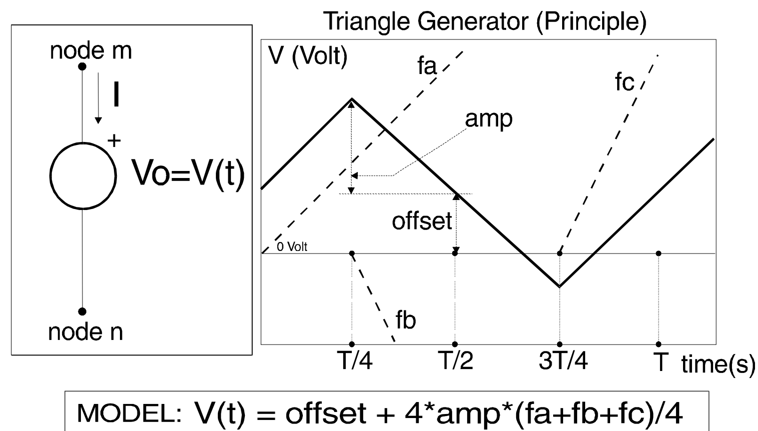
Circuit description:

```
#include <alec.h>
#define Period 15 ms
root eq ()
{
  MyResistor r;
  Sinus singen;

  r(1,0)1k;
  singen (1,0) {amp=1V; freq=200Hz; phase=0.5rad; offset=3V;}

  timing { tstop = Period; a_step=Period/500; }
  plot { caption "User Defined Sinus
                Generator";
        node 1; }
}
```

Simulation results:



A6.1.12. Triangular voltage generator



Model description:

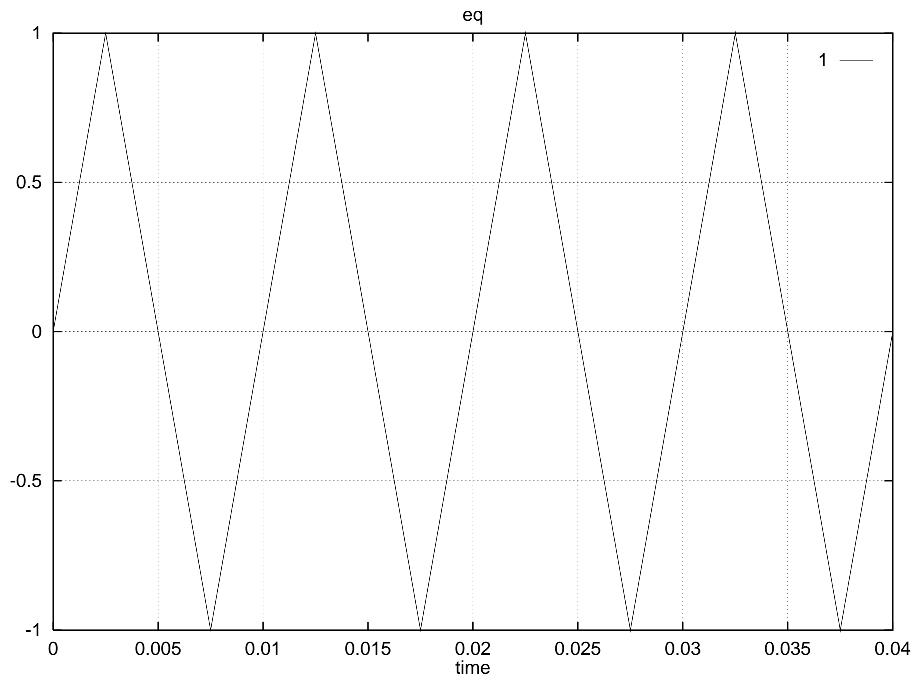
```
module current Triangle (node m, n) {
  vgen Triangle;
  return Triangle(m, n);   // connection definition

  action per_moment ( double amp=1.0V,
                      double T=1s,
                      double offset=0V) {
    // control parameter
    double X;
    double fa=0, fb=0, fc=0;

    X = now;
    while(X > T) X -= T;

    fa = X;

    if (X>T/4) fb=-2*(X-T/4);
     else fb=0;

    if (X> 3*T/4) fc=2*(X-(3*T/4));
    else fc=0;

    Triangle->value = offset + 4*amp*(fa+fb+fc)/T;
  }
}
```

A6.1.13. Example 5

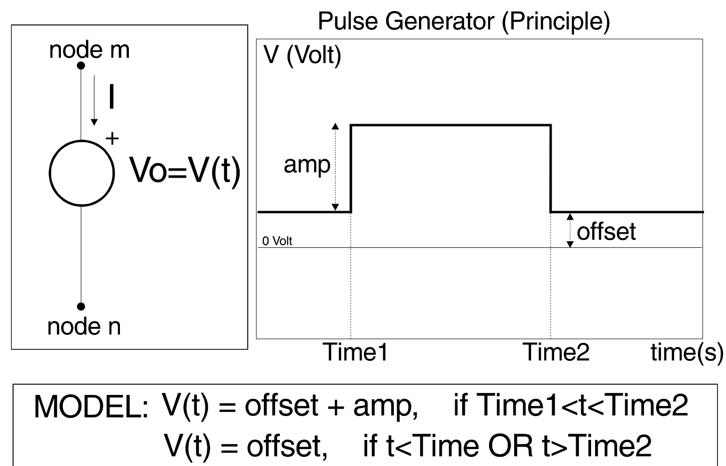
Circuit description:

```
#include <alec.h>
#define Period 40 ms

root eq ()
{
   MyResistor r;
   Triangle t;


  r(1,0)1k;
  t(1,0) {amp=1V; T=10ms; offset=0V;}

  timing { tstop = Period; a_step=Period/10000; }
  plot { node 1; }
}
```

Simulation results:



A6.1.14. Pulse voltage generator



Model description:

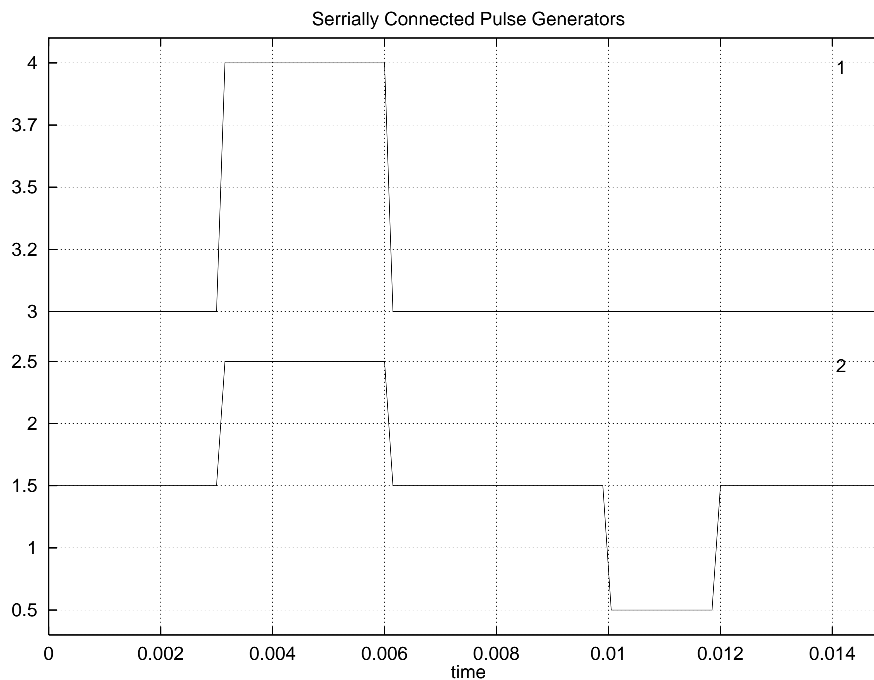
```
module current Pulse (node m, n) {
   vgen Pulse;
   return Pulse(m, n);
   // default parameters definition
   action per_moment (double amp=1.0V,
                      double time1=1ms,
                      double time2=2ms,
                      double offset=0V) {
      if ((now>=time1) && (now<time2))
      Pulse->value = amp+offset;

      if ((now<time1) || (now>=time2))
      Pulse->value = offset;
   }
}
```

A6.1.15. Example 6

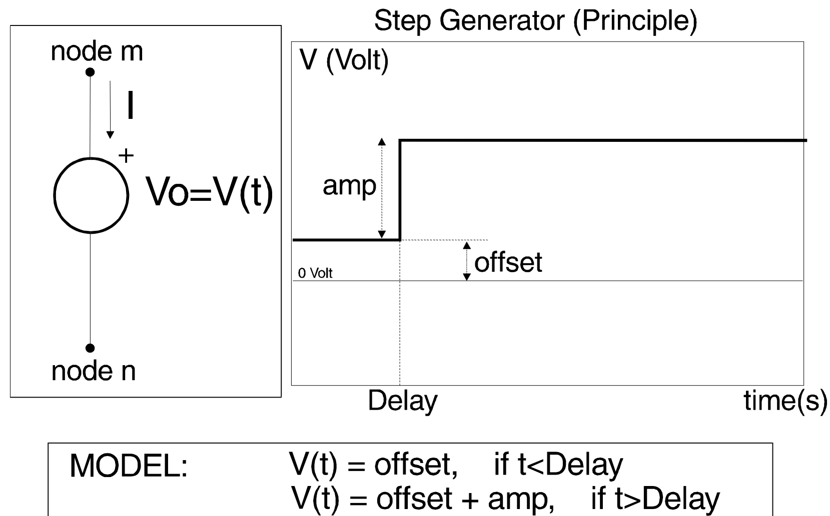
Circuit description:

```
#include <alec.h>
#define Period 15 ms

root eq ()
{
   MyResistor r;
   Pulse p1, p2;

   r(2,0)1k;
   // two generators produce independant,
   // but superimposed pulses
   p1 (1,0) {amp=1V; time1=3ms; time2=6ms; offset=3V;};
   p2 (1,2) {amp=1V; time1=10ms; time2=12ms; offset=1.5V;};

   timing { tstop = Period; t_step=Period/100; }
   plot { caption  "Serrially Connected Pulse Generators";
       node 1;  node 2; }
}
```

Simulation results:



A6.1.16. Step voltage generator



MODEL: V(t) = offset, if t<Delay
V(t) = offset + amp, if t>Delay

Model description:

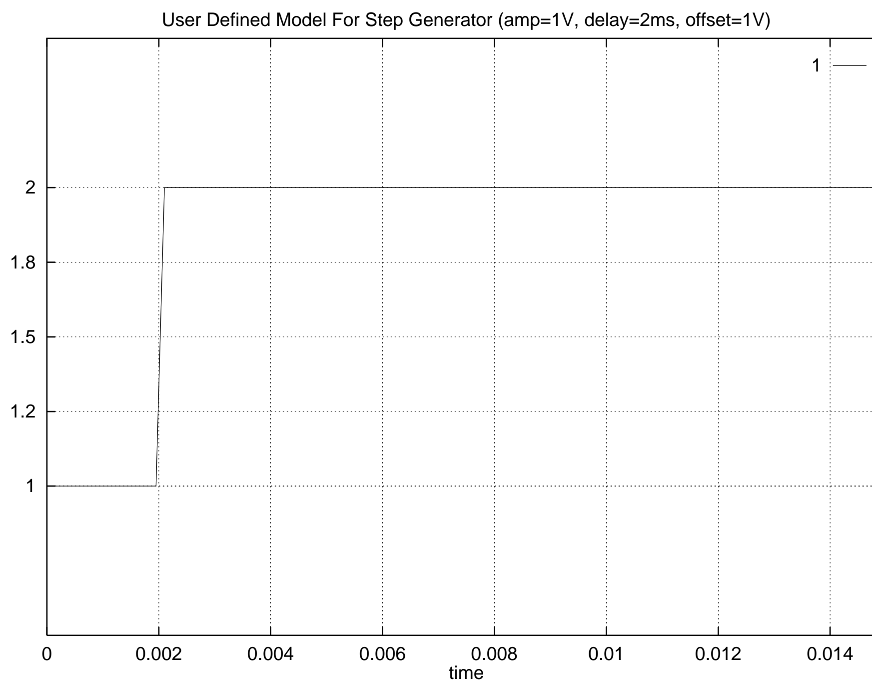
```
module current Step(node m, n) {
 vgen Step;
 return Step (m,n);    // node definition

 action per_moment (double amp=1.0V,
                    double delay=1ms,
                    double offset = 0V) {
    if (now < delay)
       Step->value = offset;
    else
       Step->value = offset + amp;
    }
}
```

A6.1.17. Example 7

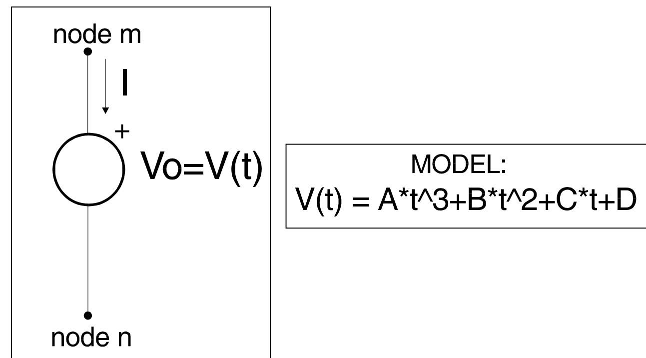
Circuit description:

```
#include <alec.h>
#define Period 15 ms

root module eq () {
  MyResistor r;
  Step sgen;

  r (1,0)1k;
  sgen (1,0) {amp=1V; delay=2ms; offset=1V;};

  timing { tstop = Period; a_step=Period/100; }
  plot { caption "User Defined Model For \
                Step Generator (amp=1V, delay=2ms, offset=1V)";
         node 1; }
}
```

Simulation results:



A6.1.18. Polynomial voltage generator



Model description:

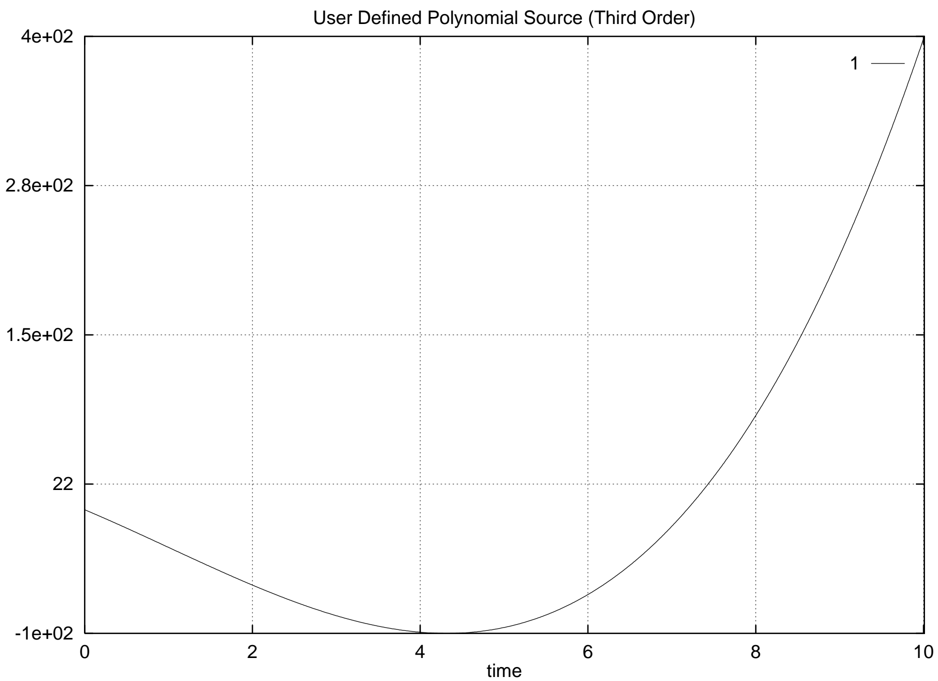
```
module current Poly (node m, n) {
   vgen Poly;

   return Poly (m, n);

   action per_moment (double A=0.0,
                      double B=0.0,
                      double C=0.0,
                      double D=0.0) {
      Poly->value = A*now*now*now+B*now*now+C*now+D;
   }
}
```

A6.1.19. Example 8

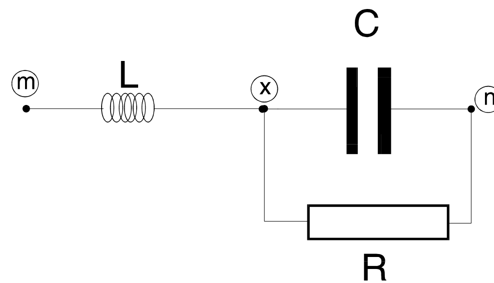
Circuit description:

```
#include <alec.h>
#define Period 10. s

root eq () {
  MyResistor r;
  Poly pgen;

  r (1,0) 1k;
  pgen (1,0) {A=1; B=-3; C=-30; D=0; }

  timing { tstop = Period; a_step=Period/1000; }
  plot { caption "User Defined Polynomial
           Source (Third Order)";
         node 1; }
}
```

Simulation results:



A6.1.20. Submodule (subcircuit) example



Submodule is described as a module without functional part (`action` block):

```
module coilcap (node m, n) {
  node x;

  resistor R;
  capacitor C;
  coil L;

  R(x,n) 1k;
  L(m,x) 1uH;
  C(x,n) 50pF;
}
```

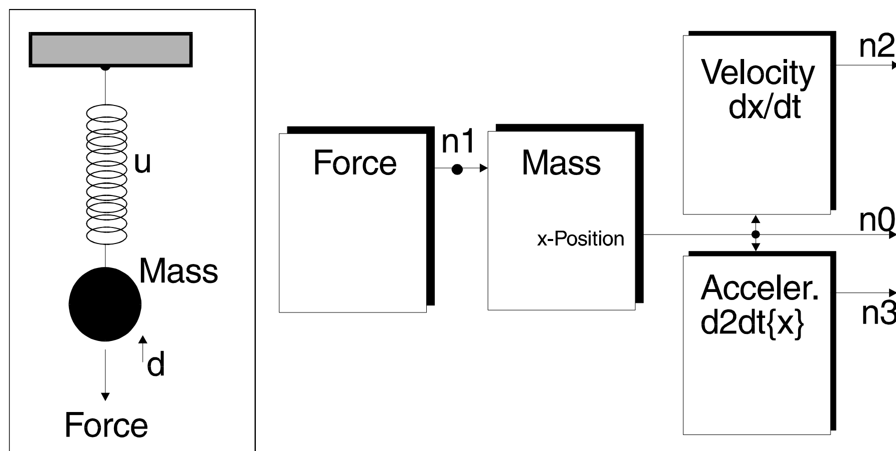
A6.2. Mechanical models

Description of mechanical models does not differ significantly from models of electronic components. One should use analogue links of type `flow` (instead of `node`, `current` and `charge` in electronics), which allows separate control of tolerance parameters for nonelectrical links.

A6.2.1. Oscillating mass

Model:

$$m\frac{d^2x}{dt^2}+d\frac{dx}{dt}+ux=F(t)$$



Model description: Nonelectrical quantites are declared as `flows`. Modules for calculating velocity and acceleration are not necessary for simulation, they are used just to print out those results:

```
// Swinging Mass
module Swinging_Mass (flow x, force) {
  action (double m, double u, double d) {
    process per_moment {
      // equation of mechanical equilibrium
      eqn x: m*d2dt2{x} +d*ddt{x} + u{x} - {force} = 0;
    }
  }
}

// Force
module Force (flow force) {
  action (double force_value) {
    double force_out;
    process per_moment {
      force_out = force_value*exp(-now);
      eqn force: {force} = force_out;
```

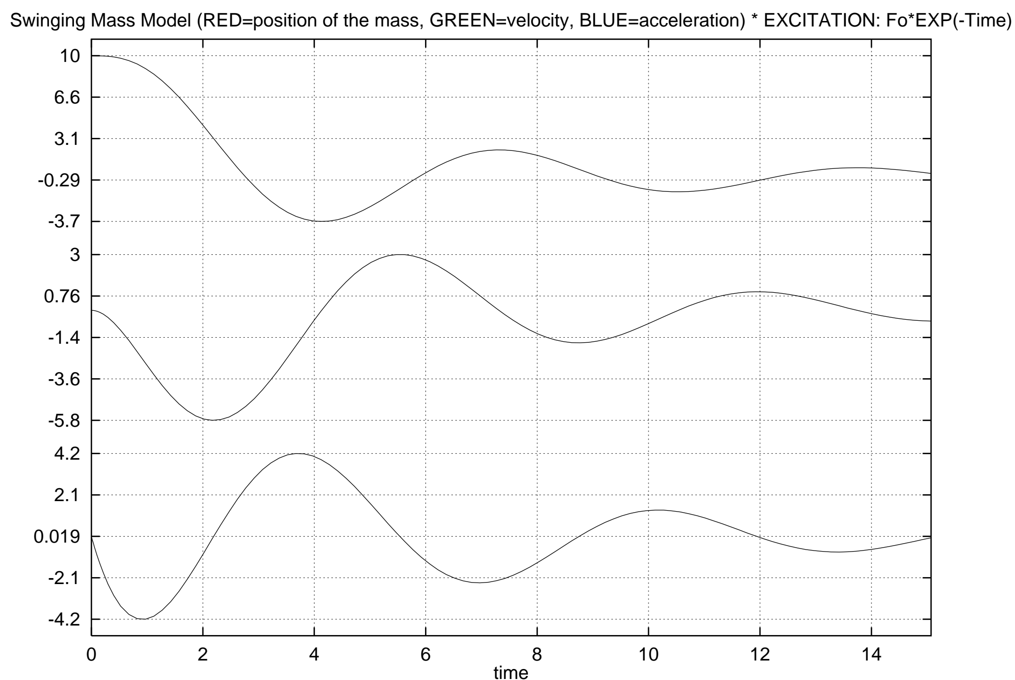
```
      }
    }
  }

  // Velocity (used for printing out, not necessary for simulation)
  module Velocity (flow x, velocity) {
    action {
      process per_moment {
        eqn velocity: {velocity} - ddt{x} = 0;
      }
    }
  }

  // Acceler. (used for printing out, not necessary for simulation)
  module Acceleration (flow x, acceleration) {
    action {
      process per_moment {
        eqn acceleration: {acceleration} - d2dt2{x} = 0;
      }
    }
  }
```

A6.1.2. Example 1

System description:

```
#include <alec.h>
#define Period 15. s

root eq () {
  flow n0, n1, n2, n3;
  // simple mechanical system
  Swinging_Mass p;
  Force F;
  Velocity V;
  Acceleration A;

  p (n1,n0){m=1 ; u=1 ; d=0.35;}
  F (n0){force_value=10;}
  V (n1,n2);
  A (n1,n3);

  timing { tstop = Period; a_step=Period/1000; }
  plot { caption "Swinging Mass Model
       (RED=position of the mass,
       GREEN=velocity,
       BLUE=acceleration)
       * EXCITATION: Fo*EXP(-Time)";
       flow n1; flow n2; flow n3;
       }
}
```

Simulation results:



Appendix 7
Alecsis library manager -- alm

While speaking about Alecsis we mentioned forming of libraries. Alecsis supports use of libraries, and we recommend it. All function calls, as well as calls of modules, model cards, and other object can be performed from previously formed libraries. Library file can store the following objects: global data (**`data`**), functions (**`funct`**), model cards (**`model`**), and modules (**`module`**). The user can define data, functions, model cards, and modules, than compile them and create a library. This also means that you can create a commercial library containing, for example models of some widely used logic circuits in a particular technology, than compile it, and distribute it without the source code.

If a user works with multiple libraries concerning a particular field of simulation, it is not very efficient for him or her to always list the libraries to be searched during the linking process. It is also not practical to expect the user to remember which module is in which library. It would be useful if a user could create the most suitable library by freely manipulating the elements of existing libraries (these are: listing the content, copying of elements, erasing, moving, listing, and similar operations with the elements of a library).

A7.1. Handling of libraries in Alecsis

Let us say that the file named `file.ac` contains something we wish to compile and store in a library file. We can obtain that library file by calling Alecsis using option '`-c`', which points out the file to be compiled but not linked.

```
alec -c file.ac
```

The result of this command is the creation of a library file named `file.ao`.

Library content can be listed using the option '`-a`' while giving the full name of the library file (including the extension '`.ao`')

```
alec -a file.ao
```

In case `root module` is in the file `file.ac`, we will receive a warning and this module will not be compiled into the library file. The `root module` can be compiled and linked only during a simulation, since it cannot be invoked from another `module`.

We can use command `library` (can be anywhere in the simulation file) to instruct compiler which libraries to search during the linking procedure.

```
library file, other_library_1, other_library_2;
```

The other possibility to transfer the information to linker is to list the libraries at the command line using option '`-l`':

```
alec -lfile -lother_library_1 -lother_library_2 test_file.ac
```

With the command `library` you can list the whole absolute or relative path in quotations to the library file. To avoid this you define the path to directory where Alecsis should search for libraries by setting the environment variable `ALEC_LIB_PATH`, or you can list these directories from the command line using option '`-L`', as described in *Appendix 1*.

You can see that program Alecsis does not have extensive capabilities in handling libraries. These capabilities are limited to listing of libraries, which is not enough when dealing with complex library files. For manipulating libraries, you can use program for library processing named **`alm`** (Alecsis Library Manager). We will discuss this program in the following sections.

A7.2. Capabilities of alm

`alm` should provide easy manipulation with the elements of libraries. The program works interactively, or it can execute a series of commands from a batch file. When started interactively program gives a prompt:

```
1:alm ->
```

At this moment the user types the commands, and after every command the command counter increases by one. When the user opens a library, the name of the library its name is written in the prompt instead the name of the program `alm`. Command **`open`** opens a desired number of libraries, as many as the operating system can handle. The current library is called the **foreground library**. All other libraries are **background libraries**. A newly opened library becomes the foreground library, except if the changes to the previous foreground library are not saved. The change of status of a library command is performed using:

```
fg lib_name
```

This command makes the previously opened library `lib_name` the foreground library, and the library is ready to be processed. However, if there are changes in the content of the previous foreground library, the user, before the change of the foreground library has to save the changes, or withdraw from the changes. This points out the program organization: **changes can be applied to one library at a time**, only. The other libraries can be listed, read, but the changes are possible only with the foreground library. All changes are stored in the memory, and the library does not change really until saving of the changed library is requested.

Command **`ll`** gives the list of libraries that are open.

Command **`ls`** lists the content of a foreground or explicitly named library. Command **`ls`** with the option **`-l`** gives detailed information about the content of a library.

Command **`h`** gives a short description of all `alm` commands.

Command **`close`** closes the library. You cannot close the library whose content is copied into the foreground library, until saving the new content of the foreground library, or giving up the changes.

Command **`alias`** defines alias (abbreviations) for more complex library names, or strings of commands. For example, command:

```
alias ttl /users/hybrid/alec/exams/lib/ttl.ao
```

enables use of shorter name of the absolute path to the library `ttl.ao` in the directory `/users/hybrid/alec/exams/lib`. This command can define aliases for other `alm` commands, too. For example:

```
alias dir ls -l
```

If your command has more than one option, you can list these options separately in any order, or you can combine them. All option arguments begin with a hyphen '-'. The following commands are equivalent:

```
ls -l -n12 lib_name.ao
ls -ln12 lib_name.ao
ls -n12 -l lib_name.ao
```

Interactive approach is satisfactory for the largest number of applications. There are, however, applications when a repeated or a packet processing of commands is needed. One application is with the copyingof libraries in parts, when some parts of libraries, depending upon

the user, should be installed in a resulting library, and some should not. In this case you can call a program to execute commands from a file. However, this way of work is not flexible, since the user cannot take the appropriate action in case of an illegal command, that is the program stops working giving out the error message. To overcome this obstacle you can use option '`-v`' that prints the current command on the screen, so you can follow the execution.

You can invoke UNIX commands from `alm`. Everything following the exclamation sign '`!`' will be passed to the operating system. This can be used for an overview of the directory content, copying a file, change of the status of a library, etc.

Notice that `alm` honours the rights of access imposed by the operating system. This means that a user cannot change a library file if he or she has the read-only authorization. If the user wants to change such library, he or she can save it under a different name, or change its status using appropriate commands of the operating system.

Program `alm` occupies less than 100Kb of RAM memory, without dynamically allocated memory for data storage when working with libraries. The program is in C according to the old definition by B.W. Kernighan and D.M. Ritchie, and is made for UNIX environment.

A7.3. Command line options

Program is invoked using command `alm`.

Syntax:

`alm [-i] [-v] [filename]`

Remarks:

When `alm` uses commands from batch file, it stops working whenever it reaches an instruction requiring interaction with the user, and gives an error report.

Options:

`-i` starts work from the named batch file. `alm` reads, and executes line after line. Every line beginning with '`#`' is a non executable command (comment).

`-v` when working from a batch file, it writes out the current command.

A7.4. alm commands

A7.4.1. List of alm commands

alias	creating aliases
close	closes a library file
cp	copying the elements into the foreground library
fg	sets the foreground library
h	prints the list of commands
ll	printing list of open libraries
ls	lists content of the named library
new	creates a new library file of the given name
open	opens a library file
q	ends the program
rep	moving elements inside the foreground library
rm	deleting elements from the foreground library
touch	updating the last modification date
w	saving a library
!	call of a shell command

A7.4.2. Overview of alm commands

alias

Effect:

Creates an alias.

Syntax:

```
alias [new_name] [old_name]
```

Remarks:

Command without arguments shows the already defined abbreviations. All abbreviations can be redefined. It is not possible to create an abbreviation for the command `alias`.

close

Effect:

This command closes a previously opened library.

Syntax:

```
close lib_name[.ao]
```

Remarks:

The extension '`.ao`' is not necessary in the name of the library. It is not possible to close a library whose elements are copied to the foreground library before recording the foreground library. When you close the foreground library, the first opened library becomes the foreground library.

`cp` (copy)

Effect:

Copies elements from a named library to the foreground library.

Syntax:

`cp [-n`*`num`*`] [-t] lib_name[.ao]/element_name`

Remarks:

It is legal to use meta-characters '`*`' and '`?`' with the elements to be copied (see command `touch`). The extension '`.ao`' is not necessary in the library name. It is not legal for two elements to exist in the same library under the same name, so the elements that already exist in the foreground library are not copied.

Options:

`-n`*num* — *num* is a positive integer. The first copied element comes to position *num*; the second on the position *num*+1 and so on.

`-t` — default option, copy to the end of the library.

`fg` (foreground)

Effect:

Previously opened library, whose name is the argument of the command, becomes the foreground library. The modifications can be performed in the foreground library only.

Syntax:

```
fg lib_name[.ao]
```

Remarks:

The extension '`.ao`' is not necessary in the name of the library. If the changes to the foreground library are not saved, you cannot declare another library the foreground library. The program will offer you to save the changes to the foreground library, give up the changes to the foreground library, or give up the whole action.

h (help)

Effect:

This command gives a list of `alm` commands along with the syntax.

Syntax:

```
h
```

ll

Effect:

Gives the list of all open libraries.

Syntax:

```
ll
```

ls

Effect:

Shows the content of a library.

Syntax:

`ls [-l] [-n`*num*`] [-s] [lib_name[.ao]] [element_name]`

Remarks:

The extension '`.ao`' is not necessary in the name of the library. If the name of the library `lib_name` is not given, default is the foreground library. You can use meta-characters (see command `touch`) in `lib_name` i `element_name`.

Options:

`-l` gives additional information on elements. Class of element is given beside the name of the element (`module`, `model`, `data`, `funct`), as well as the position of the element, the length, and the date of the last change.

-n*num* *num* is a positive integer. This option defines the number of lines printed on the screen as *num*, when the system waits for the user to press *return* for the printout to continue.

-s goes along with setting of the parameter `element_name`. This option lists only the element with the name `element_name`, if it exists in the library. The option is useful if the library has a large number of elements, and you want to find the one with a particular name.

Examples:

`ls` lists all elements of the foreground library

`ls lib_name` lists all elements of the library `lib_name` (it has to be open)

`ls -s inv_model` checks if the foreground library has the element `inv_model`

`ls -s inv*` lists all elements of the foreground library whose name begins with `inv`

`new`

Effect:

Creates a new library file that initially does not contain any objects. You can write desired content into it by copying from other libraries.

Syntax:

```
new lib_name[.ao]
```

Remarks:

The extension '`.ao`' is not necessary in the name of the library. A newly opened library becomes the foreground library, except if the changes to the previous foreground library are not saved.

`open`

Effect:

This command opens the library and reads the heading into `alm`. This procedure allows the access to the elements of the library (`alm` cannot access a library that is not open).

Syntax:

```
open lib_name[.ao]
```

Remarks:

The extension '`.ao`' is not necessary in the name of the library. A newly opened library becomes the foreground library, except if the changes to the previous foreground library are not saved.

q (quit)

Effect:

Exit from `alm`.

Syntax:

```
q[!]
```

Remarks:

If there are changes to the foreground library, the user can save the changes, leave without changes or cancel the action. The special form of the command '`q!`' lets the user leave the program without any checks.

rep (replace)

Effect:

Shifts an element within the foreground library.

Syntax:

```
rep sourcenum destnum
```

Remarks:

You give the order number of the element you wish to shift - `sourcenum`. The order number the element will obtain in the library is `destnum`. The elements with the number `destnum`, and all other numbers with the order number larger than `destnum` increase their number by one. If `destnum` is larger than the total number of elements in the library, the destination is the end of the library.

rm (remove)

Effect:

Deletes an element from the foreground library.

Syntax:

```
rm element_name
```

Remarks:

It is legal to use meta-characters '`*`' and '`?`' with the elements to be deleted (see command `touch`).

`touch`

Effect:

The command records a new modification date for the elements it is applied to.

Syntax:

```
touch element_name
```

Remarks:

It is legal to use meta-characters '`*`' and '`?`' with parameters `element_name`. Character '`?`' replaces any one character, and '`*`' replaces any group of characters (including an empty string).

Example:

```
touch inv?a*
```

This is: apply the date to all elements whose name starts with `inv`, than there is any character, then `a`, and than anything till the end of the name. For example, names satisfying this criteria are `invqa`, `invaanam`, `inv1a_model`, `inv3a`, `inv_a_____32`.

w (write)

Effect:

Records the foreground library under the new or existing name.

Syntax:

```
w [lib_name[.ao]]
```

Remarks:

The extension '`.ao`' is not necessary in the name of the library. If no name is given the library is saved under the present name, with the loss of the previous content. Otherwise, the library is saved under the new name, while the old library stays unchanged. In that case, the new library becomes the foreground library, and the old library is erased from the list of open libraries.

!

Effect:

Calls a command of the operative system, i.e. passes the string to the UNIX shell for execution.

Syntax:

```
!string
```

Examples:

```
!ls
!mkdir new_lib
```

Appendix 8
Waveform display program - agnu

Alecsis creates files with results (extension '`.ar`'), and does not create graphical representations of these results. A separate program, named **`agnu`** is used for that. Current version of this program is 1.1). Program agnu is an interface to widely distribute program **`gnuplot`**. `agnu` invokes `gnuplot`, and send simulation results in appropriate format. In other words, for `agnu` to work `gnuplot` needs to be present.

Program is invoked as:

```
agnu file_name[.ar]
```

Extension '`.ar`' can be omitted when invoking the program. All waveforms specified in **`plot`** command, i.e. printed out in file for results, will be drawn on the screen. Analogue waveforms can be grouped, i.e. can use the same scaling, or can be drawn separately. This is controlled by `plot` command. See Chapter 5 for detailed explanation of this command. Digital waveforms are always drawn separately.

After displaying the waveforms, `gnuplot` prints its prompt and stays open for interactive control. Program `agnu` accepts `gnuplot` commands and options adds some of its own, which permits a more flexible work environment.

A8.1. agnu command line options

All command line options beginning with '-' are directly passed to `gnuplot`. Options begining with '+' are new options, defined for `agnu` only:

agnu [gnuplot options] [+am] [+h[*analog_ratio*]] [+l] name_file

Options starting with '+' have the following effect:

+am without this option, y-axis range for each waveform is determined by the minimal and maximal value of the waveform. With such scaling, neighbouring waveforms can "touch" each other, making graphics unreadable. This option gives somewhat higher range on y-axis for waveforms, which creates distance between them.

+h[analog_ratio] this option defines the part of the screen occupied by analogue waveforms. Input file has to posses both analogue and digital waveforms for this option to take effect. By default, half of the screen is used for analogue waveforms and half for digital waveforms (default value for this option is '`-h`' or '`-h1`', which gives drawing proportions 1:1). Since digital values are drawn on separate waveforms, and analogue can be on the same one, this can result in a disproportionally large analogue waveforms compared to the digital. The value of parameter *analog_ratio* less than 1 reduces the analogue part, and vice_versa.

+l opens the window of large dimensions (default is small).

Explanation of `gnuplot` options can be found in the Manual for this program.

A8.2. agnu commands

When the prompt appears, you can use the following `agnu` commands beside `gnuplot` commands (gnuplot commands by typing `help`).

g on/off of the grid

z 2x enlargement with respect to the centre of the graphic

Z 2x reduction with respect to the centre of the graphic

l, r, u, d moving the window for a half of the size with the respect to the centre of the graphics (**l** - left, **r** - right, **u** - up, **d** - down).

b returns back to the original graphics regardless of the transformations

Using commands of program `gnuplot`, graphics can be exported to a file in a large number of formats (HPGL, PostScript, EPS, etc.).

Appendix 9
Postprocessors nrl and nzd

Programs **`nrl`** (no repeating lines) and **`nzd`** (no zero delay) are used to **postprocess the file with results of Alecsis digital simulation**, i.e file with extension '.ar'. You can call them in the following way:

```
nrl file_name.ar
nzd file_name.ar
```

It is necessary to type the extension '`.ar`'.

Input of these programs is the output file of the program Alecsis. The output is file with the same name - previous content is erased. If the content is to be saved, it should be copied before, under a different name. In case you forget to do this, you can always replay the simulation and regenerate the output file.

In digital simulation, results are printed whenever there is an event on traced signals. Some of these events can be **neutral events**, having no effect on the output drivers. They are printed out by Alecsis, but apart for giving information about neutral events, these lines are useless and reduce readability of the output file. Postprocessor **`nrl`** removes all repeating lines (only one copy of a line is left). **Line is considered as repeating, if all traced signals have the same value as in the previous line, and the value of the simulation time is the same as in the previous line.**

Processor **`nzd`** removes all lines with the same value of simulation time, except the last one. When a digital circuit is simulated using **zero delay**, there can be such lines in the output file. Such events cannot be viewed on the graphics anyway, as they are drawn for the same time instant.

The effects of these processors are shown on an example. Simulation output file `test.ar` has the following content:

```
adder_15_test

OutputVariables:
TIME    0:x[0] 0:x[1] 0:x[2] 0:s 0:c_out 0:ncout

OutputValues:

0.0000e+00  0  0  0  *  *  *
0.0000e+00  0  0  0  *  *  1
0.0000e+00  0  0  0  *  0  1
0.0000e+00  0  0  0  *  0  1
0.0000e+00  0  0  0  0  0  1
1.0000e+00  0  0  0  0  0  1
2.1000e+00  0  0  1  0  0  1
2.1000e+00  0  0  1  0  0  1
2.1000e+00  0  0  1  1  0  1
3.0000e+00  0  1  0  1  0  1
3.0000e+00  0  1  0  1  0  1
3.0000e+00  0  1  0  0  0  1
3.0000e+00  0  1  0  0  0  1
3.0000e+00  0  1  0  0  0  1
3.0000e+00  0  1  0  1  0  1
4.0000e+00  0  1  1  1  0  1
4.0000e+00  0  1  1  1  0  1
4.0000e+00  0  1  1  0  0  0
4.0000e+00  0  1  1  0  1  0
5.0000e+00  1  0  0  0  1  0
5.0000e+00  1  0  0  0  1  0
5.0000e+00  1  0  0  1  1  1
5.0000e+00  1  0  0  1  0  1
```

After execution of the command:

`nrl test.ar`

file `test.ar` has the following content:

```
adder_15_test

OutputVariables:

TIME    0:x[0] 0:x[1] 0:x[2] 0:s 0:c_out 0:ncout

OutputValues:

0.0000e+00  0  0  0  *  *  *
0.0000e+00  0  0  0  *  *  1
0.0000e+00  0  0  0  *  0  1
```

```
0.0000e+00  0  0  0  0  0  1
1.0000e+00  0  0  0  0  0  1
2.1000e+00  0  0  1  0  0  1
2.1000e+00  0  0  1  1  0  1
3.0000e+00  0  1  0  1  0  1
3.0000e+00  0  1  0  0  0  1
3.0000e+00  0  1  0  1  0  1
4.0000e+00  0  1  1  1  0  1
4.0000e+00  0  1  1  0  0  0
4.0000e+00  0  1  1  0  1  0
5.0000e+00  1  0  0  0  1  0
5.0000e+00  1  0  0  1  1  1
5.0000e+00  1  0  0  1  0  1
```

After execution of the command:

```
nzd test.ar
```

file `test.ar` has the following content:

```
adder_15_test

OutputVariables:

TIME   0:x[0] 0:x[1] 0:x[2] 0:s 0:c_out 0:ncout

OutputValues:

0.0000e+00  0  0  0  0  0  1
1.0000e+00  0  0  0  0  0  1
2.1000e+00  0  0  1  1  0  1
3.0000e+00  0  1  0  1  0  1
4.0000e+00  0  1  1  0  1  0
5.0000e+00  1  0  0  1  0  1
```

Appendix 10
PSpice to Alecsis converter

A10.1. Why PSpice2Alecsis conversion?

PSpice is an integrated mixed-mode (hybrid) simulator of the electronic circuit. Therefore, it can be used for analogue, digital and hybrid circuits. The input language of the PSpice simulation is based on the input language of SPICE simulator complemented by the mechanisms for digital circuit description.

PSpice2Alecsis is a program that converts PSpice description of hybrid electronic circuit into the equivalent description in AleC++. The execution version of PSpice2Alecsis program is called `p2a`.

There are three main reasons for development of this program:

- Providing a possibility to compare simulation results of the same circuit obtained by PSpice and Alecsis.
- Possibility to use big number of already developed and available PSpice libraries in Alecsis.

- **PSpice2Alecsis enables PSpice/Alecsis co-simulation**. It means that one part of the circuit is described by using components and commands of the input language of PSpice, while the other part of the circuit is described by using AleC++ constructions and principles.

Why is PSpice/Alecsis co-simulation needed? PSpice is a language for electronic circuit description, but without a possibility to describe new models. AleC++ possesses mechanisms for modelling of new components. Besides, systems that are not of electrical nature can be described in AleC++ and simulated in Alecsis, such as micromechanical systems, computer networks, neural networks etc. Therefore, if a complex system includes standard electronic components, that are well characterized in PSpice, but also components that cannot be found in PSpice, PSpice/Alecsis co-simulation can be the best solution.

Note: PSpice2Alecsis is under active development. Therefore, in the moment when you are reading this, new features may be added already. Besides, some of the limitations are due to Alecsis, and not the `p2a` converter. However, Alecsis is going to be improved to allow better compatibility with PSpice, too.

A10.2. Use of PSpice2Alecsis

The program is invoked from the command line as:

```
p2a file_name.ext
```

Where '`.ext`' is extension for PSpice input file (usually '`.cir`'.). This extension has to be specified when invoking `p2a`, if it exists in the name of the PSpice input file. As a result, two output files are created:

```
file_name.ac
file_name.cmm
```

AleC++ description is stored in `file_name.ac`, while comments on the conversion are stored `file_name.cmm`. These comments include PSpice command list from `file_name.ext`, which do not have its equivalent command (construction) in AleC++, nor can they be realised in AleC++ in some other way, as well as some commands whose conversion has not been implemented yet. In some specific situations, it can also contain some instructions for PSpice description, which may lead to better conversion to AleC++. If the conversion is successful, file for comments `file_name.cmm` is empty.

In `file_name.ext`, commands for including other files (or their parts) can be used:

```
.inc "inc_file_name.ext"
.lib "lib_file_name.ext"
```

can be used. In such case, conversion will be performed for these files, too. Therefore, files `inc_file_name.ac` , `inc_file_name.cmm`, `lib_file_file.ac`, `lib_file_name.cmm`, are also created.

In the beginning of each file that is a product of `p2a` program, following information is given: version of the program, date and time of current file generation, input file name, output file name with extension '`.ac`' and output file name with extension '`.cmm`'.

One of the major parts of PSpice2Alecsis converter is the parser, which identifies input language commands of PSpice simulator. Depending on which command is identified, an equivalent AleC++ command or language construction is generated. Two groups of commands could be identified in input language of PSpice simulator:

- commands describing circuit topology (components and their connections);
- commands for simulation control.

Commands for circuit topology description start with a letter (the first letter of particular component name), whereas commands for flow control analysis start with a point '`.`'.

PSpice identifies 22 different type of elements. They are not declared, since the first letter in the component name denotes the type of the component. There are three types of components:

- ♦ analogue,
- ♦ digital (whose name starts with `U`)
- ♦ A/D and D/A converters for hybrid circuit (`N`, `O`)

A10.2.1. Analogue circuit conversion

PSpice2Alecsis converter reads the description in PSpice and creates description in AleC++. Alecsis equivalents (AleC++ modules) of PSpice analogue components are already prepared in the library, which is named `PSpicecomp`. This library is already compiled, and is automatically included in resulting file using `library` command. Therefore, in each AleC++ source `file_name.ac` that is created by `p2a`, following commands are found:

```
#include "RLC_model.h"
#include "PSpicecomp.h"
library PSpicecomp;
```

It is supposed that these files are in directory visible to Alecsis compiler (see *Appendix 1* on Alecsis installation and usage). Files with extension '`.h`' are appropriate header files, containing necessary declarations for definitions stored in the library.

In `PSpicecomp` library, there are the following modules, which correspond to certain components from input language of PSpice simulator:

`resistorPSp`	resistor (**`R`***name* component)
`capacitorPSp`	capacitor (**`C`***name* component)
`inductorPSp`	inductor (**`L`***name* component)
`vsingen`	voltage source for a sinusoidal waveform which returns branch current on its name

`vsingenv`	voltage source for a sinusoidal waveform
`vexpgen`	voltage source for an exponential waveform which returns branch current on its name
`vexpgenv`	voltage source for an exponential waveform
`vsffmgen`	voltage source for a frequency-modulated waveform which returns branch current on its name
`vsffmgenv`	voltage source for a frequency -modulated waveform
`vpulgen`	voltage source for a pulse waveform which returns branch current on its name
`vpulgenv`	voltage source for a pulse waveform
`csingen`	current source for a sinusoidal waveform which returns (branch) current on its name
`csingen0`	current source for a sinusoidal waveform
`cexpgen`	current source for an exponential waveform which returns (branch) current on its name
`cexpgen0`	current source for an exponential waveform
`csffmgen`	current source for a frequency-modulated waveform which returns (branch) current on its name
`csffmgen0`	current source for a frequency-modulated waveform
`cpulgen`	current source for a pulse waveform which returns (branch) current in its name
`cpulgen0`	current source for a pulse waveform
`ccswPSp`	current-controlled switch (**`W`***name* component)
`vcswPSp`	voltage-controlled switch (**`S`***name* component)

In case there are more files containing circuit descriptions, these files are included in the resulting AleC++ file containing `root module`.

A10.2.2. Digital circuit conversion

All digital components are sorted as one component type in PSpice, and their name starts with `U`. Models of digital components that are implemented in PSPICE are prepared in libraries. There are three libraries, and two files containing necessary declarations and definitions:

`PSpicedef.h`	header file that contains used structure declarations, global data, functions and modules
`PSpicefun`	library that contains functions used in modelling (late functions, resolution functions etc.)
`PSpicestr`	library that contains digital `module` definition
`PSpicemod`	library that contains redefined model cards

`.alecrc` header file that contains certain variable definitions

These five files ought to be visible by Alecsis, and are included in AleC++ file created by `p2a`:

```
#include "PSpicedef.h"
#include ".alecrc"
library PSpicemod, PSpicesr, PSpicefun;
```

In case there are more files containing circuit descriptions, these files are included in the resulting AleC++ file containing `root module`.

A10.2.3. Hybrid circuit conversion

Hybrid simulation means that a described circuit contains both analogue and digital components. Due to different nature of mechanisms for analogue and digital simulation, analogue and digital domains need to be divided by A/D and D/A converter insertion. Converter insertion is executed by PSpice simulator itself (the same is valid for Alecsis, too). It can be also done manually, by the user. A converter that is inserted is a subcircuit, which contains **O***name* or **N***name* component for A/D or D/A converter, respectively.

A10.3. Conversion of PSpice commands with examples

A10.3.1. Conversion of components

C*name* command:

This command is used to specify a capacitor in the input language of PSpice simulator.

Example 1:

In PSpice input language, capacitor named `c1` is defined using:

```
c1   11 12 3.498E-12 ;
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
root ...// or module
{
  // declartion part
  capacitor c1;
  ...
  // structure part
```

```
  c1 (11, 12) 3.498e-12;
  ...
}
```

Example 2:

In PSpice input language, capacitor named `c14` and its model card `capmodel` are defined as:

```
c14 21 22  capmodel 300nF IC = 2.0V
.model capmodel cap (vc1=1.0 vc2=2.0 tc1=3.0 tc2=4.0)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include "pspcomp.h"
library "pspcomp";

//model card for new component capacitorPSp
model CPSp::capmodel{
   vc1=1.0;
   vc2=2.0;
   tc1=3.0;
   tc2=4.0;
}
root ... // or module
{
  // declaration part
  capacitorPSp c14;
  ...
  // structure part
  c14 (20, 21)  { model = cmodel; value = 300; }
  ...
}
```

`R`*name* command:

This command is used to specify a resistor in input language of PSpice simulator.

Example 1:

In PSpice input language, resistor named `r705` is defined using:

```
r705 c1 c2  55
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
root ... // or module
{
  // declaration part
  resistor r705;
  ...
  // structure part
```

```
  r705 (c1, c2)  { value = 55; }
  ...
}
```

Example 2:

In PSpice input language, resistor named `r604` and its `TC` parameter are defined using:

```
r604 13 10  24.87 TC=10,1
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include "pspcomp.h"
library "pspcomp";

//model card for new component resistorPSp
model RPSp::TCr604{
   tc1=10;
   tc2=1;
}

root ... // or module
{
  // declaration part
  resistorPSp r603;
  ...
  // structure part
  r603 (13, 10)  { model = resmod; value = 24.87; }
  ...
}
```

Example 3:

In PSpice input language an resistor named `r603`, its `TC` parameter and its model card `resmod` are defined as:

```
r603 13 10 resmod  24.87 TC=10,1
* command for specifying model card
model resmod res (r=1.5 tc1=0.02 tc2=0.005
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
include "pspcomp.h"
library "pspcomp";

// model card for new component resistorPSp
model RPSp::resmod{
   r=1.5;
   tc1=0.02;
   tc2=0.005;
}

root ... // or module
{
  // declaration part
```

```
  resistorPSp r603;
  ...
  // structure part
  r603 (13, 10)  { model = resmod; value = 24.87; }
}
```

D*name* command:

This command is used to specify a diode in input language of PSpice simulator.

Example:

In PSpice input language diode named `d12` and its model card are defined as:

```
d12 c31 c32 dmodel AREA = 20.9
.model dmodel D (Is=1E-13 Vj = 0.7)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
spice {
.model dmodel d ( is=1e-13 vj=0.7 ) }
root ... // or module
{
  // declaration part
  diode d12;
  ...
  // structure part
  d12 (c31, c32) { model = dmodel; area = 20.9; }
  ...
}
```

G*name* command with specification `poly`:

In the input language of PSpice simulator, this command is used to specify a voltage-controlled current source (with polynomial dependence).

Example:

In PSpice input language, an voltage-controlled current source named `g983` which has three pairs of nodes for voltage control, and eight coefficients, is defined as:

```
g983 983 0 poly(3)(1 2)(3 4)(5 6) 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
module gpoly3 (node i,j, n1, n2, n3, n4, n5, n6)
{
   cgen genc;
   genc (i,j);
```

```
   action per_moment (double p0=0.0, double p1=0.0, double p2=0.0,
double p3=0.0, double p4=0.0, double p5=0.0, double p6=0.0, double
p7=0.0, double p8=0.0, double p9=0.0)

   {
      genc->value = p0 + p1*(n1-n2) + p2*(n3-n4) + p3*(n5-n6) +
p4*(n1-n2)*(n1-n2) + p5*(n1-n2)*(n3-n4) + p6*(n1-n2)*(n5-n6) +
p7*(n3-n4)*(n3-n4) + p8*(n3-n4)*(n5-n6) + p9*(n5-n6)*(n5-n6);
   }
}
root ...{ // or module
  // declaration part
  gpoly3 g983;
  ...
  // structure part
  g983 (983, 0, 1, 2, 3, 4, 5, 6) {p0=1.0; p1=2.0; p2=3.0; p3=4.0;
  p4=5.0; p5=6.0; p6=7.0; }
}
```

Note 1:

`Poly` specification of `E`*name* and `G`*name* commands has two syntax options:

with brackets:

```
G983 983 0 poly(3) (1 2)(3 4)(5 6)  1.0 2.0 3.0 4.0 5.0 6.0 7.0
```

without brackets:

```
G983 983 0 poly(3)  1 2  3 4  5 6   1.0 2.0 3.0 4.0 5.0 6.0 7.0
```

Note 2:

Value of the controlling variable, having polynomial dependence on V_1, V_2, …, V_n, is defined :

$$
\begin{aligned}
V_{out}(V_1, V_2, \ldots, V_n) = \; & P_0 + \\
& P_1 * V_1 + P_2 * V_2 + \ldots + P_n * V_n + \\
& P_{n+1} * V_1^{\,2} + P_{n+2} * V_1 * V_2 + \ldots + P_{n+n} * V_1 * V_n + \\
& P_{2n+1} * V_2 * V_2 + P_{2n+2} * V_2 * V_3 + \ldots + P_{2n+n-1} * V_2 * V_n + \\
& \ldots + \\
& P_{n!/(2(n-2)!+2n)} * V_n * V_n
\end{aligned}
$$

Where: $P_0, P_1, P_2, \ldots$ are polynomial coefficients.

`E`*name* command with specification `value:`

This command is used in input language of PSpice simulator to specify a voltage source, whose value is controlled by a function.

Example:

In PSpice input language, a function that controls voltage source named `esum` is defined using following command:

```
esum 4  0  value = {v(2)*v(6)*i(vr)*v(5)*i(vg)}
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
module  vvalue1 (node i,j; node n1; node n2; current c1; node n3;
current c2)
{
   vgen genv;
   genv (i,j);

   action per_moment ()
   {
      genv->value =  n1*n2*c1*n3*c2;
   }
}

root ... // or module
{
  // declaration part
  vvalue1 esum;
  ...
  // structure part
  esum (4, 0, 2, 6, vr, 5, vg);
  ...
}
```

`E`*name* command with specification `table`:

This command is used to specify a voltage source, whose control is given as a table.

Example:

In PSpice input language, voltage source named `erele` whose value is controlled by a function, is defined with the following command:

```
erele  2  0  table {V(1)} = (2, -1) (2.01, 1)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
module current vtable1 (node i,j; node n1)
{
   vgen vtable1;
   return vtable1 (i,j) 0.0;

   action per_moment ()
   {
     int loop;
     int i, j;
```

```
        double izlaz, x1, x2, y1, y2;
        static const double table[2][2] = {
            2, -1,
            2.01, 1,
        };

        loop = 1;
        i = 0;
        while (loop == 1)
        {
           if ( n1 > table[i][0])
           {
             if (i < 1)
               i++;
             else
             {
               loop = 0;
               izlaz = table[1][1];
             }
          }
           else
           {
             loop = 0;
             if (i != 0)
             {
                x1 = table[i-1][0];
                x2 = table[i][0];
                y1 = table[i-1][1];
                y2 = table[i][1];
                izlaz = (( n1 - x1)/(x2 - x1))*(y2 - y1) + y1;
             }
             else if( n1 < table[i][0])
                izlaz = table[i][1];
           }
        }
        vtable1->value = izlaz;

      }
   }

   root ... // or module
   {
     // declaration part
     vtable1 erele;
     ...
     // structure part
     erele (2, 0, 1);
     ...
   }
```

`V`*name* (`I`*name*) command

`V`*name* command is used to specify an independent voltage source in the input language of PSpice simulator. `I`*name* command is used to specify an independent voltage source in the input language of PSpice simulator.

Example:

In PSpice input language an independent voltage source with sinusoidal waveform named `vsin is` defined with the following command:

```
vsin 10 5 sin(2 2 5Hz 1 10 30)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include "pspcomp.h"
library "pspcomp";
root ... // or module
{
  // declaration part
  vsingen vsin;
  ...
  // structure part
 vsin (10,5) { voff=2.0v; vampl=4.0v; freq=50hz; td=1msec; df=10;
 phase=30; }
  ...
}
```

`X`*name* command:

In the input language of PSpice simulator, this command is used to specify call of a subcircuit.

Example:

In PSpice input language, call of subcircuit named `xcomp` is defined with using following command (called subcircuit has five nodes (`0`, `3`, `nvdd`, `nvss, 4`) and subcircuit's definition has name `tl064/ti`):

```
xcomp  0  3  nvdd  nvss  4  tl064/ti
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
root ... // or module
{
  // declaration part
  tl064_slash_ti xcomp;
  ...
  // structure part
  xcomp (0, 3, nvdd, nvss, 4);
}
```

`M`*name* command:

This command is used to specify a MOSFET in the input language of PSpice simulator.

Example:

In PSpice input language, MOSFET named `mmosfet` and its model card `modmos` are defined as:

```
mmosfet   0   2   100   100   modmos L=33u W=12u
+ AD=288p AS=288p PD=60u PS=60u NRD=14 NRS=24 NRG=10
.model modmos nmos ( lambda=2 )
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
spice {
.model modmos nmos ( lambda=2 ) }
root ... // or module
{
  // declaration part
  mosfet mmosfet;
  ...
  // structure part
  mmosfet (0, 2, 100, 100) {model = nweak; l=33u; w=12u; ad=288p;
  as=288p; pd=60u; ps=60u; nrd=14; nrs=24; nrg=10;}
  ...
}
```

`Q`*name* command:

This command is used to specify a BJT in the input language of PSpice simulator.

Example:

In PSpice input language, BJT named `mmosfet` and its model card `modbjt` are defined as:

```
qbjt   a   b   c   model605   area=24.87
.model modbjt npn (Is=17.01E-12 Bf=110 Vje=0.85)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
spice {
.model modbjt npn ( is=17.01e-12 bf=110 vje=0.85 ) }
root ... // or module
{
  // declaration part
  bjt qbjt;
  ...
  // structure part
  qbjt (a, b, c) { model = model604; area = 24.87;  }
  ...
}
```

`J`*name* command:

This command is used to specify a JFET in the input language of PSpice simulator.

Example:

In PSpice input language, JFET named `jjfet` and its model card `modjfet` are defined as:

```
Jjfet  a  b  c  modjfet  area=24.87
.model modjfet pjf(Is=17.01E-12 Beta=110.5E-6 Vto=-1)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
spice {
.model modjfet pjf ( is=17.01e-12 beta=110.5e-6 vto=1 ) }
root ... // or module
{
  // declaration part
  junctionfet jjfet;
  ...
  // structure part
  jjfet (a, b, c) { model = model603; area = 24.87; }
}
```

`S`*name* command:

This command is used to specify a voltage-controlled switch in the input language of PSpice simulator.

Example:

In PSpice input language, voltage controlled switch named `svcs` and its model card `sw1mod` are defined as:

```
svcs 13 17 2 0 sw1mod
.model sw1mod vswitch (ron=1.0 roff=1e+6 von=1.0 voff=0.0)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include <alec.h>
#include "pspcomp.h"
library "pspcomp";

// Model card for new component vcswPSp.Coressponds to the .model
// in example.
model vSWPSp::sw1mod{
   ron=1.0;
   roff=1e+6;
   von=1.0;
   voff=0.0;
}

root ... // or module
{
  // declaration part
  vcswPSp svcs;
```

```
  // structure part
  ...
  svcs (13, 17, 2, 0)  model = sw1mod;
  ...
}
```

`W`*name* command:

This command is used to specify a current-controlled switch in the input language of PSpice simulator.

Example:

In PSpice input language, current-controlled switch named `wccs` and its model card `wmod` are defined as:

```
wccs 13 17 vc wmod
.model wmod iswitch (ron=2.0 roff=1e+9 ion=1e-2 ioff=1e-6)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include <alec.h>
#include "pspcomp.h"
library "pspcomp";

// Model card for new component AleC++ ccswPSp. Coressponds to the
//.model in example.
model cSWPSp::wmod{
   ron=2.0;
   roff=1e+9;
   ion=1e-2;
   ioff=1e-6;
}

root ... // or module
{
  // declaration part
  ccswPSp wccs;
  ...
  // structure part
  wccs (13, 17, vc)  model = wmod;
}
```

`U`*name* command:

This command is used to specify digital components in the input language of PSpice simulator.

Example 1:

In PSpice input language, an AND logical circuit named `uand21`, and its model cards `io1` (input/output model card) and `tm01` (timing model card), are defined as:

```
.model io1 UIO (inld=0.1 outld=0.2 drvh=1.0 drvl=2.0 drvz=3.0)
.model tm01 UADC (tphlmn=1 tphlty=2 tphlmx=3 tplhmn=0.1 tplhty=0.2
tplhmx=0.3)
uand21 AND(2) $G_DPWR $G_DGND in0 in1 out tm01 io1
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include "pscdef.h"
# include ".alecrc"
library pscmod, pscstr, pscfun;

model uadc::tm01_io1{
   tphlmn=1.0
   tphlty=2.0
   tphlmx=3.0
   tplhmn=0.1;
   tplhty=0.2;
   tplhmx=0.3;
   inld=0.1;
   outld=0.2;
   drvh=1.0;
   drvl=2.0;
   drvz=3.0;
}

root ...  // or module
{
  module ugate::and_2 uand21;
  ...
  uand21(out,in0,in1) model = tm01_io1;
  ...
}
```

Example 2:

In PSpice input language, a digital stimulus generator with `LOOP` specification named `ustim` is defined with the following command.

```
Ustim STIM(4,13)
+ $G_DPWR $G_DGND
+ 4 3 2 1 IO_STIM5 TIMESTEP=10ns
+ 0c 00
+ 5c 03
+ LABEL=STARTLOOP1
+ 100ns decr by 01
+ 200ns goto startloop1  until lt 00
+ +10ns  13
+ 700ns  06
+ LABEL=STARTLOOP2
+ 720ns  07
+ 800ns  10
```

```
+ 900ns  goto startloop2 2 times
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include "pscdef.h"
#include ".alecrc"

library pscmod, pscstr, pscfun;
module STIM1ustim (signal ps_full out out1, out2, out3, out4)
{
   signal four_full y[1:4];
   action ()
   {
       process
       {
         int init = 1;

         if (init) { y <- "0000"; init = 0; wait y; }
         else
           y <- "0011" after 50ns; wait y;
           y <- "0010" after 50ns; wait y;
           y <- "0001" after 100ns; wait y;
           y <- "1011" after 110ns; wait y;
           y <- "0110" after 390ns; wait y;
           y <- "0111" after 20ns; wait y;
           y <- "1000" after 80ns; wait y;
           y <- "0111" after 100ns; wait y;
           y <- "1000" after 80ns; wait y;
           y <- "0111" after 100ns; wait y;
           y <- "1000" after 80ns; wait y;

       } //process

       process {
         int init = 1;
         if (init)
         {
            out1 <- '0';
            out2 <- '0';
            out3 <- '0';
            out4 <- '0';
            init = 0;
            wait y;
         }

         out1 <- y[1];
         out2 <- y[2];
         out3 <- y[3];
         out4 <- y[4];
         wait y;
       } // process

   } //action
}


root ... // or module
{
```

```
  module STIM1ustim uname;
  ...
  ustim (4, 3, 2, 1) { model = io_stim5;}
  ...
}
```

A10.3.2. Conversion of simulation control statements

.TRAN statement:

The `.tran` statement causes a transient analysis to be performed. The general form of the statement is:

```
.tran[/OP] <print_step> <final_time> [no_print [step_celling]] [UIC]
```

The transient analysis calculates the circuit's behaviour over time, starting at TIME = 0 and going to `<final time>`. Alecsis, in its current release, performs transient analysis only, so this command can be realized in AleC++.

`OP` specification demands printing of the complete information about DC analysis in textual output file. This has not its equivalent in AleC++.

`UIC` specification orders simulator to set the voltage across the capacitors and the current through the inductors at DC analysis, which is used to determine limit conditions for transient analysis. Since Alecsis in this release does not possess a mechanism for setting the voltage across the capacitors and the current through the inductors, `UIC` specification cannot be realised either.

AleC++ equivalent for PSpice `.tran` command is is `timing` command. As we have already explained, `timing` command does not support all parameters supported by `.tran` command. `.tran` command parameters supported by `timing` command are:

print_step --- *tprint* (Alecsis)

final_time --- *tstop* (Alecsis)

step _celling --- *a_stepmax* (Alecsis)

`timing` command does not support *no_print* parameter. However, `timing` command demands a parameter which does not exist in `.tran` command - *a_step* parameter. It is calculated for `timing` command on the basis of *print_step* parameter from `.tran` command, as *print_step* divided by 5.

Example 1:

In PSpice input language `tran` statement in basic form (with two parameters that cannot be omitted) is defined as.

```
.tran 2u 10m UIC
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
timing { tprint = 2u; a_step = 2u/5; tstop = 10m;  }
```

Example 2:

In PSpice input language `.tran` statement in complete form (with all parameters) is defined as:

```
.tran  2u  20m  1u  2u  UIC
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
timing { tprint = 2u; a_step = 2u/5; tstop = 20m; a_stepmax = 2u;}
```

`.TEMP` statement:

The `.temp` statement sets the temperature at which simulation is performed. If more than one temperature is given, then simulation is repeated for each temperature. The general form of `.temp` statement is:

```
.temp <temperature_value>*
```

The equivalent command to `.temp` command from PSpice in Alecsis is `temp` option in `options` command. The limitation of Alecsis is that it does not possess a mechanism to execute simulation for more than one temperature. Because of that, if more than one temperature value appears in `.temp` command PSpice2Alecsis will take only the first one, and in the file for commenting conversion (a file with '`.cmm`' extension) a note will be found that `.temp` command does not have a fully equivalent command in Alecsis.

An example of `.temp` command conversion into `. options` command follows. It should be noted that temperature in PSpice is specified in Centigrade degrees, while in Alecsis it is in Kelvin degrees.

Example:

In PSpice input language temperature is defined using the following command:

```
.temp 50 75
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
options {temp = 323.000; }
```

`.SUBCKT` statement:

This statement begins the definition of a subcircuit. The definition is ended with `.ends` statement. All the statements between `.subckt` and `.ends` form the subcircuit the definition.

Subcircuit definition statements should contain only topology description (statements without a leading '`.`'), and possibly `.model` statements.

The general form of `.subckts` statement and complete form of subcircuit are :

```
.subckt <name_subcircuit> [node]*
       [optional : <<interface_node> = <default_value>>*]
       [params : <<name> = <value>>*]
       [text : <<name> = <text> = <value>>*]
       ; structure block
 .ends [name_subcircuit]
```

A subcircuit from PSpice corresponds to Alecsis `module`.

Note:

`optional` and `text` specifications are not realised in converter.

Example:

In PSpice input language, beginning of a subcircuit named `ICL7652/TI` is defined using the following command.

```
.subckt ICL7652/TI 1 2 3 4 5
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
module icl7652_slash_ti  (node 1; node 2; node 3; node 4; node5) {
...//structure of module (subcircuit)
}
```

`.PRINT` and `.PROBE` statements:

The `.print` statement prints out results from dc, ac, noise, or transient analysis in the form of table, referred to as print tables. The `.print/dgtlchg` form is for digital output variables only. The general form of the statement is:

```
.print [/DGTLCHG][DC][AC][NOISE][TRAN][(output_variable)]*
```

The `.probe` statement writes the results from dc, ac, and transient analyses to a file `probe.dat` for use by the *Probe* graphic postprocessor. The general form of the statement is:

```
.probe [/csdf] [output_variable]*
```

Note:

For showing simulation results in PSpice we can use either `.print` command or `.probe` command, or both simultaneously. Both commands can be used without arguments, and in that case complete simulation results are included (values of all circuit quantities). (Conversion into AleC++ code is not supported for this case.). `.print` and `.probe` commands could print a great number of output value categories. In this release of PSpice2Alecsis, conversion of only one category - node voltage (e.g. `V(1)`, `V(a_node)`) - is supported.

Example:

In PSpice input language, `.PRINT` and `.PROBE` statements are defined using following two statements.

```
.print tran v(305) v(505) v(105 505) v(308 108) v(1408 708)
.probe v(608) v(1349 321) v(1402) v(305) v(505) i(lname) i(vname)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
plot {
       //PRINT output :
       node 305; node 505;
       //PROBE output :
       node 608; node 1402; node 305; node 505;
}
```

`.PARAM` statement:

This command defines global parameters of simulation. A global parameter can be a constant or expression. This command is realised in AleC++ by means of preprocessor command `#define`.

Example:

In PSpice input language a `.param` statement is defined with the following command.

```
.param e19 = {1 / (6.28 * sqrt(ll * cc)) }
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#define e19 1/((6.28*sqrt(ll*cc))
```

`.INC` statement:

The `.inc` statement is used to insert (include) the content of another file into the current file. Including files is the same as simply bringing the file's text into the current file. Included files may contain all statements with these exceptions: no title lines is allowed (use a comment); `.end` statement is not allowed. The general form of `.inc` statement is:

```
.inc "file_name"
```

Note:

The current converter realisation allows that only subcircuits and `.inc` command can be found in the included file.

Example:

In PSpice input language, `.inc` statement is defined with as.

```
.inc "dat1.mod"
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include "dat1.ac"
```

`.LIB` statement:

The `.lib` statement is used to reference a model or subcircuit library in another file. The convenience (with respect to `.inc` command) is that the complete library is not read through, but only needed objects are found and included in circuit description.

The general form of the statement is:

```
.lib "file_name"
```

Note:

For the time being, `.lib` statement is realised in the same way as `.inc` command - like `#include` preprocessor command.

Example:

In PSpice input language `.lib` statement is defined as:

```
.lib "dat1.mod"
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include "dat1.ac"
```

`.FUNC` statement:

This statement is used to define "functions" that may be used in expressions. This command is realised in AleC++ like the macro by using `#define` preprocessor command.

Example:

In PSpice input language `max` functions is defined using the following command.

```
.func max (a,b) (a + b + abs(a - b))) / 2
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#define max (a,b) ((a)+(b)+abs((a)-(b))))/2.0
```

A10.4. Limitations of PSpice2Alecsis converter

This chapter deals with the restrictions of PSpice2Alecsis program. There are two causes for restrictions:

- some commands and specifications of the simulator input language can not be realised in AleC++, due to differences in simulator engines.
- conversion for certain commands and command specifications of PSpice simulator input language is not yet implemented in PSpice2Alecsis

Limitations are divided into three groups:

- limitations of commands describing circuit topology (components and their connections);
- limitations of commands for simulation control;
- general limitations.

In the next sections, following description is used:

yes - PSpice command is completely supported

no - PSpice commands is only recognised, but not supported

yes* - PSpice command is partly supported

A10.4.1. Limitations of circuit topology description

`B`*name* --- no
`C`*name* --- yes* (`IC` specification is not supported.)
`D`*name* --- yes
`E`*name* --- yes* (`FREQ` specification is not supported.
`LAPLACE` specification is supported only under the following conditions:

- transfer function can be : $\frac{A*s+B}{C*s^2+D*s+E}$,
- A, B, C, and D coefficients can constants or variables, but more complex expressions are not allowed.

– expression which shows control input can only be voltage or current.)

F*name* --- yes

G*name* --- yes* (FREQ specification is not supported.

LAPLACE specification is supported only under the following conditions:

– transfer function can be : $\frac{A*s+B}{C*s^2+D*s+E}$,

– A, B, C, and D coefficients can constants or variables, but more complex expressions are not allowed.

– expression which shows control input can only be voltage or current.)

H*name* --- yes

I*name* --- yes* (AC and DC specifications are not supported.)

J*name* --- yes

K*name* --- no

L*name* --- yes* (IC specification is not supported.)

M*name* --- yes

N*name* --- yes

O*name* --- yes

Q*name* --- yes

R*name* --- yes

S*name* --- yes

T*name* --- no

U*name* --- yes* (MNTYMXDLY and IO_LEVEL specification are not supported.)

V*name* --- yes* (AC and DC specifications are not supported.)

W*name* --- yes

X*name* --- yes* (TEXT specification is not supported.)

A10.4.2. Limitations of commands for simulation control

.ac --- no

.dc --- no

.distribution --- no

.end --- yes

.ends --- yes

.four --- no

.func --- yes

.ic --- yes* (Assignment of voltage between two nodes is not supported, because this version of Alecsis does not possess the convenient mechanism. For the same reason, IC specifications in commands for inductor and capacitor description are not supported.)

.inc --- yes* (Included library, according to the PSpice syntax, can contain all PSpice commands except title lines and .end command. However, in an included file, PSpice2Alecsis supports only commands that can be found in the subcircuit (commands for description of PSpice components and/or subcircuits) and .inc command.)

.lib --- yes* (For the time being, .lib command is realised the same way as .inc command - as #include preprocessor command.)

`.loadbias` --- no
`.mc` --- no
`.model` --- yes
`.nodeset` --- no
`.noise` --- no
`.op` --- no

`.options` --- yes* (Only `temperature` parameter is supported.)
`.param` --- yes
`.plot` --- no

.print --- yes* (See example for `.print` command in the section A10.3.2.)
`.probe` --- yes* (See example for `.probe` command in the section A10.3.2.)
`.savebias` --- no
`.sens` --- no
`.step` --- no

`.subckt` --- yes* (`OPTIONAL` and `TEXT` specification are not supported.)
`.temp` --- yes* (More than one temperature value is not supported, because Alecsis does not possess the convenient mechanism to repeat simulation.)
`.tf` --- no

`.tran` --- yes* (`no_print` parameter is not supported,
`UIC` specification is not supported.

`.watch` --- no
`.wcase` --- no
`.width` --- no
`.text` --- no

A10.4.3. General limitations

- At conversion, capital letters become small ones (small letters stay the same).
- In AleC++, characters '`$`', '`%`', '`*`', and '`/`' cannot be used in names of variables and nodes. For that reason, when PSpice2Alecsis reads such signs, it converts them into '`_dollar_`', '`_percentage_`', '`_slash_`', and '`_star_`', respectively.
- Voltage between two nodes cannot be initialised in this version of Alecsis (only node votlage with respect to ground node can be initialzied). It means that the command

```
.ic v(1, 2) = 5.0v.
```

  cannot be realised in `IC` specification in `L`*name* and `C`*name* commands.
- Connection of text lines using '+' sign is not made possible for every command. For instance, in `.subckt` command, it is not supported that each node name can be found in separate line. The same applies to `STIM` specification of `U`*name* command.
- PSpice2Alecsis does not recognise line comment in lines that are the continuation of the previous ones (beginning with '+')

- PSpice2Alecsis does not allow usage of following keywords as names of variables and nodes: `TIMSTEP`, `MNTYMXLY` and `IO_LEVEL`
- PSpice2Alecsis does not allow usage of keyword `FILE` as the name of a variable or a node in `X`*name* and `.subckt` commands.

Alecsis Manual

Version 2.4

Release Note 1

software version 2.4.1
june 1998

Laboratory for Electronic Design Automation
University of Niš, Faculty of Electronic Engineering
Beogradska 14, 18000 Niš, Yugoslavia

2/1998

Alecsis Manual
Version 2.4
Release Note 1

software version 2.4.1
june 1998

Vladimir Risojević

Željko Mrčarica

Michel Lenczner*

Laboratory for Electronic Design Automation
University of Niš, Faculty of Electronic Engineering
Beogradska 14, 18000 Niš, Yugoslavia
http://leda.elfak.ni.ac.yu/

**Laboratoire de Calcul Scientifique*
Groupe Materiaux Intelligents
16 Route de Gray, 25000 Besançon

Note:

Introduction of new system of equations solver is explained in this release note. This solver is intended for very large matrices. It enables solution of the matrix block by block, where block which are not currently used are temporarily stored on disk. The matrix must be organized in block for this method to be effective. This is actually intended for simulation of micromechanical problems.

Three sections of the original Manual are given here. Section 5.6.3.3. replaces old version,; section A1.2.1. replaces part of the old version (section is long, so unchaned parts are not repeated here); and section A1.3. replaces old version.

5.6.3.3. Control of system of equations solver

(*new version of the section*)

Table 5.4. Control of sparse matrix solver.

Name	Default value	Meaning
`renum`	`Best (2)`	Sparse matrix renumeration algorithm. It can be `None (0)`, `Fast (1)`, `Best (2)`, or `Frontal (3)`.
`pivot_threshold`	`0`	Used if `renum` equals `Frontal`. This parameter is used by the frontal LU solver during the pivoting. It can be between 0 and 1.
`buffersize`	`0`	Used if `renum` equals `Frontal`. Size of the buffers where coefficients and its indices obtained during the frontal LU decomposition are stored before they are written to temporary files.

Sparse matrix solver can be controlled by means of the option `renum`. This option offers the possibility of choosing between standard, column-oriented, sparse matrix storing scheme (when `renum` is either `None`, `Fast`, or `Best`), and frontal scheme (`renum` is `Frontal`) which is appropriate for matrices arising from finite element method applications, and very closely related to the LU decomposition itself. Furthermore, by its value an algorithm for reordering of matrix rows and columns is specified. In detail, the number of nonzero elements in the system matrix generated during LU decomposition, and consequently memory space needed for storing calculated coefficients, depends on the ordering of matrix rows and columns. This choice can also affect the CPU time necessary for simulation. When frontal LU decomposition is chosen the parameters `pivot_threshold` and `buffersize` are used, otherwise their value is ignored.

In the case of choosing column-oriented sparse matrix storing scheme reordering is performed only once, at the beginning of simulation. If you chose option `Best`, a variant of Berry's algorithm is used, when very detailed (and slow) reordering is performed. This is the default value, as reordering is performed only once, and good reordering guaranties fast simulation. With option `Fast`, a variant of Markowitz's algorithm is used, when reordering is performed much faster, with somewhat slower simulation in time domain afterwards. This option should be chosen for very large matrices (several hundreds of equations or more), since with Berry's algorithm, reordering can take more CPU time than time-domain simulation. When option `None` is chosen, no reordering is performed. This is implemented for comparison only, it has no practical effect, since simulation can take too much time.

If option `Frontal` is chosen, a system matrix is decomposed using the frontal technique, first devised by Irons. This technique is originally developed for solving large positive-definite, symmetric sparse systems of equations such as occur in finite element method applicatons. Duff

extended frontal technique for solving arbitrary large sparse systems of equations, and, a variant of Duff's algorithm is implemented here. Option `Frontal` decomposes matrix in blocks, and can swap currently unused blocks to disk. This enables very large matrices to be solved, that cannot be held in computer memory as a whole. However, if the matrix cannot be organized in blocks, i.e. if there are many nonzero entries far away from the main diagonal, this method cannot be efficient, as blocks cannot be identified.

The basic idea of this algorithm is the fact that a system matrix is formed by performing subsequent assemblies of elemental matrices (stamps). It is obvious that there is an elemental matrix after whose assembly there are no further contributions to some row and column. We say that the variable corresponding to that row and column is fully summed, and can be eliminated if some numerical stability criterion is satisfied. After the elimination is done that row and column are removed from the matrix and the obtained factors are written to disk. In this way, the complete system matrix is never held in the main memory. Instead, all operations are performed in a matrix, called frontal matrix, whose rows and columns correspond to variables that have not yet been eliminated but occur in at least one of the elements that have been assembled. Here, the process of LU decomposition is interleaved with the assembly of system matrix, so this technique permits solving of large sparse systems of equations.

The above mentioned numerical stability criterion depends on the value of the parameter `pivot_threshold`. If its value is 0 (default) then the only constraint on the pivot value is that it must not be zero. This helps in avoid zero pivots that can arise from electrical elements like ideal voltage generators and inductors. However, it is usually important to have larger pivots, as the numerical error is smaller in such case.If the value of `pivot_threshold` is different from zero then the pivot is choosen using the following threshold pivoting criterion: a_{lk} is good pivot if

$$|a_{lk}| \geq \texttt{pivot_threshold} \cdot \max_i |a_{ik}| \qquad (5.14)$$

is satisfied. Note that `pivot_threshold=1.0` corresponds to partial pivoting.

The factors obtained during the elimination are not immediately written to disk. They are first put to buffers in the main memory and only when some buffer is full, its contents is flushed to disk. There are three buffers, two for factors (L and U matrices), and one for their row and column indices. Obviously, if the buffers are large enough it is possible to avoid the usage of disk, or at least reduce it, because it slows down the simulation. The size of the buffers can be specified by setting the value of parameter `buffersize`. It is the size of each of the three buffers. Default value is zero meaning that there is no buffering.

Note**:** The order of assembly of the elemental matrices determines the order of elimination, and therefore the memory space needed for LU decomposition, so as the simulation time. One may conclude that frontal method is not appropriate for arbitrary, large sparse matrices, but only for those with particular sparsity patterns. Block diagonal matrices represent a class of matrices on which frontal method is applicable, and they often occur in finite element problems. In general, it is desirable that the order of the frontal matrix is as small as possible. This can be achieved by means of the element reordering. Finite element codes usually yield the sequence of elements which meets this requirement.

Note: Values of parameter `renum` - `None`, `Fast`, `Best` and Frontal are actually integer values, defined in standard Alecsis header file `alec.h`. In that file, it is defined:

```
#define None      0
#define Fast      1
#define Best      2
#define Frontal   3
```

Therefore, to use textual values of parameter `renum`, you should have file `alec.h` file included before your `root module` definition, using command:

```
# include <alec.h>
```

A1.2.1. Program call from the command line -- command options

(*section update*)

`-v`*verbose_level* Gives more information about the simulation run. There are following options:

- `-v1` tracks symbol table activity;
- `-v2` tracks intermediate code generation (operand types etc.);
- `-v3` all LEX tokens are printed out as they arrive;
- `-v4` follows voltage generator/inductor loops detection;
- `-v5` prints instructions as they are flushed;
- `-v6` tracks overloading and prototype mangling;
- `-v7` prints list of nodes;
- `-v8` follows the use of weights if option `dcon` is used;
- `-v9` follows the process of static/global initialization;
- `-v10` follows library management;
- `-v11` follows function declaration;
- `-v12` clear global symbol table before simulation;
- `-v13` tracks function prototype existence;
- `-v15` tracks class member access control;
- `-v16` follows function inline expansion;
- `-v31` prints system matrix and right-hand side vector in every iteration, as without reordering. It has no effect if option `renum` equals `Frontal`;
- `-v32` prints system matrix and rhs vector in every iteration as reordered. It has no effect if option `renum` equals `Frontal`;
- `-v33` prints **both** non-reordered and reordered system matrix and rhs vector, respectively, in every iteration. It has no effect if option `renum` equals `Frontal`;
- `-v34` prints statistics if option `renum` equals `Frontal`, otherwise has no effect, viz.
  1. system size,
  2. frontal matrix size,
  3. fully summed variables block size,
  4. number of non-zero entries in the original matrix,
  5. number of factors in both L and U matrices,
  6. number of indices stored for bookkeeping purposes.
- `-v55` turns on full logic initialization
- `-v99` changes all calls of exit() with abort() to dump core file

Note: Verbose level `55` (full logic initialization) is rather a simulation option than a verbose level, and it will be organized as such in following versions of Alecsis.

Most of these verbose levels are of interest only for us that created Alecsis, for our debugging purposes. However, there are some of them that can be very useful for Alecsis users. For instance, **verbose level `8` follows use of weight when option `dcon` for difficult convergence problems is used**. This can be very useful for setting correct values for options `max_weight`, `min_weight`, `p`, `q`, and `maxdcon`, if you are not satisfied with their default values (see Chapter 5, section on simulation `options` for details).

Verbose level `31` prints out system matrix, which can sometimes be helpful if you have problems with zero pivot (singular matrix). This is, however, useful only for small matrices, as it is very difficult to analyze large matrices.

Note: If more than one *verbose_level* is given, only the last one will take effect. For example:

```
alec -v1 -v2 file_name
```

has the same effect as:

```
alec -v2 file_name
```

A1.3. Overview of Alecsis versions

(*new version of the section*)

We use notation of Alecsis versions with tree numbers. First number denotes crucial change of Alecsis/AleC++ functionality. The second one denotes change of functionality (new feature) from the user point of view. The last number is denotes improvement (usually debugging) of existing functions.

Alecsis 1.x	input language based on C, no object-orientation;
Alecsis 2.1.1. - 2.1.50	object-oriented input language AleC++ is introduced;
Alecsis 2.2.1. - 2.2.33.	operator `d2dt2` is introduced;
Alecsis 2.3.1. - 2.3.38	through and across `eqn` statements are introduced;
Alecsis 2.4.1. - 2.4.x	new frontal method for LU decomposition of large sparse matrices is introduced.