CONTENT - integrated environment for analysis of dynamical systems
Developer's guide

V. V. Levitin

January 5, 1998

Contents

Preface

CONTENT is an open multi-platform integrated environment designed for investigating the properties of dynamical systems. It was developed by Yu.A.Kuznetsov and V.V.Levitin at Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands in 1994-1997.

The "openness" of CONTENT means its ability to support new classes of dynamical systems and new numerical algorithms without disturbing existing code and computed data.

This document describes the structure of data used by CONTENT to provide the openness. It also describes the procedures needed to add new components to CONTENT. Materials from this document are intended for developers of new numerical algorithms and those who are responsible for their incorporation into CONTENT.

It is assumed that the reader is familar with CONTENT and its basic ideas.

Typographic conventions

The following typographic conventions are used in this document:

Typeface	Description
Bold	Text in bold typeface represent keywords, options, or delimiters; you must type them exactly as shown.
Italic	Text in italic typeface indicates a general kind of information (placeholder); you must supply the particular value.
Underlined	Underlined text emphasizes important terms; see the Index for all references on them.

Chapter 1

Supporting the openness of CONTENT

Special efforts were made during designing of CONTENT to make it an open environment. Technically speaking, CONTENT consists of three separate parts:

- the kernel,
- analysis procedures,
- configuration and data files.

These components are described below.

1.1 The kernel

The kernel of CONTENT constitutes an environment for analysis procedures. It does not contain any code or data specific to particular class of dynamical systems and analysis procedures. Instead, it gets various information about analysis procedures from the configuration and data files at run-time and interpret it to provide investigation of dynamical systems.

There are three logical components in the kernel:

- interface manager,
- archive manager,
- dynamic objects manager.

1.1.1 Interface manager

The interface manager performs the following tasks:

- it interacts with the user through windows and menus and allows him/her to select various objects, modify parameters of analysis procedures, and set various processing modes,
- it organizes the execution of analysis procedures (starters and generators),
- it manages various windows and visualizes computed curves.

1.1.2 Archive manager

The archive is a set of files where CONTENT stores dynamical systems with all data specified for their analysis as well as the results of the analysis. Logically, the archive consist of a number of independent units each of which contains all information related to one dynamical system.

The description of the dynamical system provided by the user includes the names of variables (e.g time, state, parameters, etc.) and C-function(s) computing values of the RHS. CONTENT creates a corresponding dynamic object and associates it with the dynamical system.

Results of investigation of the dynamical system are stored as diagrams. A diagram contains a number of computed curves. A curve contains coordinates of points and additional data required to redraw and recompute (or extend) the curve.

The archive manager provides the user with ability to store, retrieve for processing, delete, and rename dynamical systems, diagrams, and curves.

The manager maintains a map of the logical structure of the archive to file system and creates all necessary files automatically. The user does not deal with them directly.

1.1.3 Dynamic objects manager

A dynamic object is a part of CONTENT code which is not kept in the memory all the time during sessions. Instead, it is brought into the memory only when it is needed and unloaded from the memory when it completes its function.

There are following reasons to have dynamical objects:

- it may be impossible to create a part of code prior to a session (for example, a dynamical system is specified during the session and, therefore, cannot be compiled and linked in advance),
- there may be a set of 'mutually exclusive' parts of code, that is no more than one part from a group is executed at a time so including them all into the resident part would be waste of memory (for example, all starters as well as generators are mutually exclusive),
- the mechanism of dynamic loading provides 'openness' of the system, that is a possibility to add new computational components without even recompiling and relinking other parts.

The dynamic objects manager manages the following kinds of dynamical objects including compiling and linking some of them:

- functions specifying dynamical systems,
- starters with appropriate defining, test, processing functions and decoders,
- generators,
- functional parameters of starters and generators,
- visualization functions,
- utility functions (e.g linear algebra packages),
- user functions for monitoring their values or detect zeroes along curves.

CONTENT accesses each dynamic object on the ground of information obtained from the archive and configuration files.

Dynamic objects are implemented as dynamic link libraries in Windows and shared objects in Unix (except for HP versions of Unix where they are called shared libraries).

1.2 Analysis procedures

Analysis procedures form the computational core of CONTENT. They are organized as follows. For each supported class of dynamical systems and each solution type there are a set of numerical routines which compute relevant curves and perform their analysis. Each routine performs its own part of the task. These routines are stored as several dynamic objects.

1.2.1 Starters

A starter is a routine that calculates the first point of a curve from an initial point given by the user. Roughly speaking, the initial point may not satisfy the conditions defining desired curve. The starter tries to find a point on the curve.

1.2.2 Defining functions

Defining functions specify curves of particular type. They are used by the generators to compute points of curves. The total number of the defining functions is by one less than the number of coordinates of points (including active parameters, i.e. the parameters that may vary).

Usually, defining functions consist of equations specifying a phase object (for example, an equilibrium or a cycle) and equations defining its singularity (for example, the presence of eigenvalues on the imaginary axis or on the unit circle).

Defining functions may also include user functions. Appending such functions allows one to continue a singular object satisfying extra conditions. In this case, the number of active parameters must be accordingly increased.

1.2.3 Test functions

Test functions are used by a generator to detect special points on a curve. The generator monitors the values of test functions along the curve and locates their zeros. It marks a point as a special one on a basis that a certain condition involving the values of the test functions is true at the point. A typical condition is a zero value of a test function at the point.

Nomenclature of test functions and types of special points which can be detected depends on a curve type.

1.2.4 Processing functions

Processing functions are called by a generator at a special point to calculate data that characterize such point (e.g. normal form coefficients) and are needed to start another curve from this point as an initial point.

The processing functions also issue the messages about detected special points.

1.2.5 Generators

A generator computes points of a curve starting from a given first point. It also locates special points on the curve by monitoring values of test functions. It calls processing functions at special points to calculate data related to these points.

Example of generators provided by CONTENT are integrators, the iterator for iterated maps, and the continuer.

1.2.6 Adapters

An adapter is an auxiliary routine that may be called by a generator after computation of a point on a curve. It may perform some processing of internal data, e.g. rearrange mesh points.

1.2.7 Decoders

A decoder is a special routine which maps points produced by a generator into spaces visible for the user (e.g. phase space). Spaces visible for the user and these in which generators compute curves are usually different. The user does not deal with decoders. Instead, they are called internally by CONTENT when needed.

1.3 Configuration and data files

The configuration and data files are used by the kernel to obtain data related to supported classes of dynamical systems and analysis procedures.

All configuration files as well as executables and libraries must be in the same directory called the Content Main Directory.

The configuration and data files are plain ASCII text files. This allows the developer to edit existing and add new files. Users should never change configuration and data files.

It is recommended to print the contents of a configuration file before reading its description.

1.3.1 Categories of the files

The configuration and data files used by CONTENT may be grouped into several categories. Files from one category differ from files from other category by purpose, structure of data, or a way CONTENT uses them. The categories are as follows:

- kernel files contain various parameters of the kernel,
- class files describe supported classes of dynamical systems,
- curve type files describe particular types of curves for each class and analysis procedures,
- parameter files specify parameters of starters and generators visible to the user; CONTENT uses them to build windows through which the user sets and changes the parameters.
- help files contain help topics available through on-line help system.

1.3.2 Format of the files

Class, curve type and some kernel files have special so called &-format. A file in the &-format is divided into sections, each of which is headed by the section name preceded by an ampersand character, e.g. **&graph**. Symbolic references to values stored in the settings file (2.1) are allowed; they are substituted at the time the configuration file is read. A line that begins with two ampersands instructs CONTENT to read the file whose name follows the ampersands. For example, the line

&&$(Paths.KERNEL)der$(Extensions.DESC)

tells CONTENT to read the file **der** located in the directory given by the **KERNEL** key from the [**Paths**] section and with the extension given by the **DESC** key from the [**Extensions**] section of the settings file. Leading and trailing spaces are ignored. Comments begin with an exclamation mark **!**.

1.3.3 Codes, labels and v-names of objects

Some objects described in class files (chapter 3) and in curve type files (chapter 4) have three different 'names' associated with them: *code*, *label* and *v-name* (*v* stands for 'visible') . They are used for different purposes.

The *code* is an unique number 1–254 and it is used in the object description stored in the archive. For example, each dynamical system stored in the archive has the code of the class of dynamical systems it belongs to as a part of its own description.

The *label* is a short identifier used in configuration and data files for references to the object. For example, a label of a curve type defined in a class file may be used in curve type files that describe specific curve types.

The *v-name* is a text used by Content to represent the object for the user. For example, the v-name of a class of dynamical systems is shown in the Main window of Content to remind the user which class is currently selected. The *v-name* may contain spaces.

1.3.4 Using the names of variables in the files

A specification of any dynamical system contains several lists of names the user gives to variables used in the specification. For example, for ODEs Content requires the user to name state (phase) varibales, parameters and time variable. This is done in the Specification of dynamical systems dialog box opened by the **Select**|**System** and **Actions**|**New** or **Actions**|**Edit** commands.

The names are naturally grouped into classes of visible names. The developer assignes codes, labels and v-names (1.3.3) to these classes in the **&visiblenames** section of the specific class file (3.2.9).

For example, for ODEs there are three classes the user provides names for. They are identified by labels **Phase**, **Par** and **Time**.

Note that the names are not known when configuration files are prepeared; they become known when Content selects a system. At the same time the names are needed in many configuration and data files. For example, the kernel uses parameter files (chapter 5) to create windows through which the user is able to change parameter values; the names are needed to produce lines like these:

x	3.14
y	2.78
z	1.00
. . . . . .	

To solve the problem Content allows expressions of special forms be used in the configuration and data files. Let *class* be the label of a class listed in the **&visiblenames** section of the specific class file. Let $name_i$ are the names defined by the user for the *class* and d_i are dimensions of the names, $i = 0, \ldots, n-1$. Dimension of a name is defined as the number of components: it is one for simple names and the number of elements for arrays. Dimension of a class is the sum of dimensions of all the names from the class.
The following expressions have special meaning

@*class*␣**{** *any_text***}**
@*class***&**␣**{** *any_text***}**
@(*algebraic_expression***)**
and are called @-loops, @&-loops and @-expressions, respectively.

An expression **@***class*␣**{***any_text***}** is replaced by *any_text* repeated n times. Then $i-$th instance ($i = 0, \ldots, n-1$) of *any_text* is processed and following special symbols in it are replaced:

Symbol	Replaced by
$*class*	$name_i$
%*class*	i
&*class*	0 for $i = 0$, $\sum_{j=0}^{i-1} d_j$ otherwise
*****class*	'*' if d_i =1, '' otherwise
class	*class*

class may be used to postpone the replacement in inner loops.

Example. Let the names **x**, **y[3]** and **z** are defined for class **Phase**. Then the expression

@Phase␣{␣$Phase_%Phase}

is expanded to

␣x_0␣y_1␣z_2

Nested loops are allowed:

@Phase␣{@\Phase␣{$Phase_$\Phase␣}}

is first expanded to

@Phase␣{x_$Phase␣}@Phase␣{y_$Phase␣}@Phase␣{z_$Phase␣}

and then to

x_x␣x_y␣x_z␣y_x␣y_y␣y_z␣z_x␣z_y␣z_z␣

An expression **@***class***&**␣**{***any_text***}** enumerates components of the *class*, not names. The expression is replaced by a series of $d{=}\sum_{i=0}^{n-1} d_i$ copies of *any_text*. The only special symbol allowed in *any_text* is **&***class*. It is replaced by ordinal numbers of components.

There are five components in the above example: **x**, **y[0]**, **y[1]**, **y[2]**, and **z**. The expression

@Phase&␣{␣Re_&Phase␣Im_&Phase}

is expanded to

␣Re_0␣Im_0␣Re_1␣Im_1␣Re_2␣Im_2␣Re_3␣Im_3␣Re_4␣Im_4

The **%***class* symbol has different meaning when used outside @- and @&-loops. It is replaced by the dimension d of the *class*.

@-expressions are replaced by values of *algebraic_expressions*. For example, **@(%Phase*%Phase+1)** would be replaced by **26**.

1.3.5 References to functions

Some configuration and data files contain references to functions in dynamic objects. These references have the following form *object*(*function*)

where *object* is a name of dynamic object and *function* is the name of a function from the object. CONTENT uses this information to load objects and access functions. If the *object* is given by its filename only then the full name used is

$(Paths.KERNEL)***object***$(Extensions.LIB)

CONTENT gets it from the **FORMAT** key from the [**Extensions**] section of the settings file (2.1).

The form

=(*function*)

is also allowed. The = character means 'the object from the previuos line'.

Chapter 2

Kernel files

Kernel files contain various parameters of the kernel. These files are created by the setup program or supplied with CONTENT. As a rule they should not be changed.

2.1 Settings file

The file is created by the setup program and has the name **.contentrc** in Unix and **content.ini** in Windows. Each user of CONTENT has its own settings file.

The file consists of sections each of which is headed by the section name enclosed in brackets, e.g. [**Files**]. Each section contains a series of strings in the form *key*=*value*. The *key* is used to access and replace the *value*. Comment lines begin with a semicolon.

Values in the settings file may contain expressions in the form **$(***section***.***key***)**. They are called symbolic references. When CONTENT reads a value with a symbolic reference it substitutes the reference by the *value* of the *key* from the section *section*. Other configuration and data files may also use symbolic references.

Here is a description of the sections and fields found in the setting file.

[**Paths**] paths to directories used by CONTENT.

- **KERNEL** full path to the Main Content Directory.
- **KERNELW** path to the current directory.
- **INCLUDE** not used.

[**Extensions**] file extensions used on particular operating system.

- **C** source C files.
- **CPP** source C++ files.
- **H** parameter files (headers).

OBJ object modules.

RUN executable files.

BAT script/batch files.

LIB dynamic objects.

ARCH archive of dynamical systems.

TEMP work files.

SYS permanent files.

DESC class, curve type and permanent files.

DIAG diagrams.

PS PoscScript files.

ES exported systems.

ED exported diagrams.

EC exported curves.

[**Libraries**] names of dynamic objects.

RHSLIB functions specifying dynamical system.

WINLIB window functions (used along axes in graphic windows).

PARLIB functional parameters of starters and generators.

UDFLIB user functions.

WINCLASSLIB last loaded window class members.

FORMAT format used to build names of dynamic objects.

LIBNAME last loaded utility module.

AUTODIF symbolical differentiator.

AUTODIFOPT symbolical differentiation option: + means On, - means Off.

HELPLIB on-line help system.

[**Functions**] external functions in dynamical objects.

AUTODIF entry point to symbolical differentiator.

[**Files**] files used by CONTENT.

TEMP work file.

TEMP2 work file.

NAMES bit pattern of numbers used to generate names of dynamical objects.

NAMES1.2 not used.

FORMAT format used to generate names of dynamical objects.

VN internally used file.

RHS source of dynamical system functions.

WIN source of window functions.

PAR source of functional parameters.

DER source of derivatives.

UDF source of user functions.

STAGEN header included to source of functional parameters.

CLASSFILE main class file (3.1).

CLASSFILECUR current curve type file.

INITFILE CONTENT text file.

PALETTE CONTENT palette file.

[**MainWindow**] options set for the Main window.

OPTMASK individual options as set in the **Options**|**Main** window dialog box (+ means On, **-** means Off).

[**Currents**] miscellaneous current settings.

LOCK lock to prevent running the second copy of CONTENT : + during sessions, **-** otherwise.

STARTUP startup mode (**0** is standard); when set to **1** CONTENT starts with no dynamical system selected.

CLASSCUR the identifier of the selected class of dynamical systems.

DIRECTORY current directory.

ARCHIVE the name of the archive of dynamical systems.

SYSFONT currently used system font.

PRGFONT currently used program font.

W_MAIN position of the main window.

W_CURVE position of the curves dialog box.

W_POINT position of the special points dialog box.

W_HELPS position of the help search window.

W_HELP position of the main help window.

S_HELP dimensions of the main help window mesured in screen dimensions (e.g. 0.5 means half of the screen's widht or height).

[**Curves**] curves related data.

UNTITLED prefix of names of untitled curves.

UNTITLEDMAX default maximal number of untitled curves per diagram.

FILTER default archive filter.

KEYS Short-cuts used during curves computation.

MOUSE if the value is **1** CONTENT opens a window with the **Suspend**, **Resume**, **Abort** and **Switch** buttons during computation of curves; if the value is **0** the window is not shown.

EXPORTOP used to mark ordinary points when exporting curves.

[**Compiler**] data used for compilation.

INCLUDE set by setup from the 'C compiler Include' field.

LIB set by setup from the 'C compiler Libraries' field.

CPPNAME set by setup from the 'C++ compiler Name' field.

CPPLIB set by setup from the 'C++ compiler Options' field.

LOGFILE the name of log file used for redirection during compilations.

SOURCE the first parameter of the CompRhs script/batch - the name of the source file.

TARGET the second parameter of the CompRhs script/batch - the name of the target file.

DIFF the third parameter of the CompRhs script/batch - **y** requests to run symbolical differentiator.

BATCH the name of script/batch file used for Compilation (CompRhs).

MAKERHS command used to run the script/batch file.

RESPONSE the name of a file with compiler options (Windows only).

HEADERF the name of the header file included in compilation.

HEADER include directive.

HEADERPAR directives used for compilation of functional parameters.

RHSATTR attributes of functions from dynamic libraries.

CERRLINEO a message issued when bugs in declaration of variables are found.

LERRLINEI the format of linker's error messages.

CURERMSG last error message.

EXTDIFFB script/batch file user to run External Differentiator (Maple).

EXTDIFF commands put to the script/batch file for External Differentiator.

[**Visual**] data related to visualization.

WINFILE windows file.

WINHEADER the header used in compilation of window functions.

WINPAR the name of formal parameter of window functions.

WINDEFA defines used to map user names to vector parameter of window function (arrays).

WINDEFS defines used to map user names to vector parameter of window function (simple variables).

INCLUDE include directive for window functions.

HCVIEWER hardcopy previewer.

HCPRINTER hardcopy print command.

[**Params**] parameter data.

FUNPAR1 the name of first parameter of functional parameters.

PRECISION precision of numbers in parameter windows.

[**Help**] help data.

FILES help files with topic that are always available.
TEXT_FONT text font.
FORMM_FONT mathematical font.
FORMT_FONT font for text in formulas.
FORMML_FONT large mathematical font.
SCRT_FONT font for text in sub/super-scripts.
SCRM_FONT sub/super-script mathematical font.
JUMP_FONT font for hypertext links.
BOLD_FONT bold font.
PROG_FONT font for program examples.
JUMP_COLOR color of hypertext links.

[**Window_2DGraphic**] 2D graphic windows data.

HARDCOPY current hardcopy options.
NUMWIDTH width of input fields.

[**Window_3DGraphic**] 3D graphic windows data.

HARDCOPY current hardcopy options.
NUMWIDTH width of input fields.

[**Window_Numeric**] numeric windows data.

OPTIONS current options.

[**Window_ScrollTable**] numeric windows data.

OPTIONS current options.
HARDCOPY current hardcopy options.

[**KernelErrors**] contains all error messages issued by the kernel.

2.2 Text file

The file is supplied with CONTENT and has the name **init.cpf**. It contains all text strings used by the kernel: window titles, menu items, prompts, messages, etc. Kernel reads the file during each startup.

The text file has &-format . . The lines in all but two last sections have the *key=string* form. These lines should not be changed.

Two sections have special format. You may want to add lines to them.

The **&markers** section contains descriptions of markers used to mark special points. One line describes one marker. It is assumed that each marker has standard dimentions 9×9 pixels. Drawing starts from the center of the square. The description consists of drawing commands listed below. n, m and k represent digits 0–8.

Cn paints a circle of radius n.

cn frames a circle of radius n.

Sn paints a square $2n\times 2n$.

sn frames a square $2n\times 2n$.

Pn paints a polygon with vertices given by the following n commands **l**m**u**k, **l**m**d**k, **r**m**u**k, or **r**m**d**k.

pn frames a polygon with vertices given by the following n commands **l**m**u**k, **l**m**d**k, **r**m**u**k, or **r**m**d**k.

lm**u**k moves the current point m pixels left and k pixels up.

lm**d**k moves the current point m pixels left and k pixels down.

rm**u**k moves the current point m pixels right and k pixels up.

rm**d**k moves the current point m pixels right and k pixels down.

The **&psfonts** section contains names of PostScript fonts that are listed in hardcopy dialog boxes of graphic windows.

2.3 Windows file

The file is supplied with CONTENT and has the name **windows.con**. It describes common classes of result windows. Result windows specific for particular class of dynamical systems are described in **&windows** sections of the corresponding class files (3.2).

Each line of the file has the form

code library(*table*) "*Menu item*" *helpfile* "*condition*"

The *code* is an unique number 1–250 used by CONTENT to distinguish different classes of windows.

library and *table* specify the dynamic object that contains class memebers and the name of a table of pointers used by the kernel to access members (see the **window.h** file).

The *Menu item* gives the name of a command in the **Windows** menu used to open windows of the given class.

The *helpfile* is the name of the help file which describes windows of the given class.

The *condition* is an expression used to enable and disable the menu command: the command is enabled when the value if non-zero, disabled otherwise.

2.4 Palette file

The file is created by the setup program and has the name **palette.cpf**. Each user of CONTENT has its own palette file.

The file contains a seria of colors the user can choose for points and lines in graphic windows. The choice is made via appropriate attribute dialog boxes.

Each line of the file contains three numbers that specify an RGB color.

2.5 Shells file

The file is supplied with CONTENT and has the name **shells.con**. It is used by Unix version of the setup program.

Each line of the file contains two items separated by spaces:
shell_name shell_style

shell_name contains the name of a shell.

shell_style specifies the style of the shell: **k** for standard-like shells (sh, jsh, rsh, bash, ksh, etc.) or **c** for c-like shells (csh, tcsh, etc.).

2.6 Script/batch files

There are several files used by CONTENT to compile and link programs during sessions. The settings file (2.1) contains compilation related data under the [**Compiler**] section.

The script/batch file for compilation is created by the setup program. It also assigns the name of the file to the **BATCH** key. The value of the **MAKERHS** key is used to run the file. The file name is **CompRhs** in Unix and **CompRhs.bat** in Windows.

For Windows version the setup program creates also a response file that contains compiler's options and is referenced by the **CompRhs.bat** file. The name of the response file is **response.cpf**; it is written as the value of the **RESPONSE** key.

If symbolical computations of derivatives of an order higher than 3 is required, CONTENT creates a special file just before the compilation. This script/batch file has the name **CompDiff** in Unix and **CompDiff.bat** in Windows. CONTENT reads the name as the value of the **EXTDIFFB** key. The file contains commands used to run the External Differentiator. The commands are put to the **EXTDIFF** key by the setup program.

Chapter 3

Class files

Class files describe supported classes of dynamical systems. They provide the kernel with information sufficient to organize the whole process of investigating the dynamical systems from specification of the systems to computation of curves.

Each supported class of dynamical system is described by corresponding class file. In addition, the main class file refers to all individual class files and serves as starting point for access class-specific data.

All class files have the &-format.

3.1 Main class file

The main class file is supplied with CONTENT and has the name **classes.con**. CONTENT reads the name at startup from the settings file (2.1) as the value of the **CLASSFILE** key in the [**Files**] section.

The **&nodes** section lists supported classes of dynamical systems and their groups. Each line describes one class or group and has the following format:

code label m-name

The *code* is the code of the class or the group. The *label* is used in other sections for references to the class. The *m-name* is used as the menu item name represented the class or group.

The **&names** section gives each class a name (which is different from *m-name*) used in the Main window of CONTENT to represent the class. Lines have the following format:

label w-name

The **&hierarchy** section specifies the structure of the **Select**|**Class** menu in the Main window. The menu allows the user to select classes of dynamical systems. Lines may have one of the following form:

class_label class_file class_hk types_hk
group_label{
}

-

A line of the first form describes a class and a command in the **Select|Class** menu in the Main window used to choose that class. The *class_file* is the name of specific class file (3.2). If it is given by its filename only CONTENT uses the value of the **KERNEL** key from the [**Files**] section as the full path and the value of the **DESC** key from the [**Extensions**] section as the extension. The *class_hk* is the key of the help topic that describes the class. The *types_hk* is the key of the help topic that lists all supported curve types for the class. The structure of help files is given in 6.

A line of the second form describes a group of classes and a sub menu in the **Select|Class** menu.

A line of the third from marks the end of a sub menu.

A line of the fourth form creates a separator line in the menu and its sub menus.

3.2 Specific class files

There is one specific class file for each class of dynamical systems supported by CONTENT. They are supplied with CONTENT. Names of specific class files are listed in the **&hierarchy** section of the main class file (3.1).

Sections of specific class files are described below.

3.2.1 &graph section

This section specifies types of points and curves and relations between them.

The lines from the section are grouped into blocks. One block describes one type of points and lists curves that may be computed from points of the given type.

The first line of a block specifies the code, the label and the name of the points type (1.3.3). It has the following form:

point_code point_label point_name

The *point_name* is used in the **Type** menu and in the information field in the Main window. It may contain the | character that separates the name used in the **Type|Initial point** sub menu from the name used in the **Type|Curve** sub menu.

Below the line there are several (may be zero) lines that list curves possibly rooted at points of the given type. These lines have the form:

-> *curve_label curve_type_file point_hk curve_hk starter_hk generator_hk*

where *curve_label* gives the type of curves that may be computed; it must be a *point_label* from this or other block. *curve_type_file* gives the name of corresponding curve type file (chapter 4). *point_hk*, *curve_hk*, *starter_hk*, and *generator_hk* are the keys of help topics that describe the point type, the curve type, parameters of starter and generator respectively. The help file with these topics given in the **&help** section (3.2.10). The topics are accessed by the commands from the **Help** menu in the Main window.

Here is an example of a block:

```
1 O Point|Orbit
  -> O   __o  point_eq ode_or start_or  integrpar
  -> EP  __ep point_eq ode_eq start_eqp conpar
  -> LC  __lc point_eq ode_lc start_lcp conpar
```

This block defines points type with code **1**, label **O** and names **Point** (used to refer to the type of initial points) and **Orbit** (used to refer to type of points of curves). It also says that curves of three types may be computed from points of the defined type: **O**, **EP** and **LC**, where **EP** and **LC** are labels defined in other blocks. **__o**, **__ep** and **__lc** are filenames of curve type files for these three curve types. The full names CONTENT uses to access the files are

$(Paths.KERNEL)__o$(Extensions.DESC),
$(Paths.KERNEL)__ep$(Extensions.DESC),
$(Paths.KERNEL)__lc$(Extensions.DESC),
All other identifiers are help topic keys.

3.2.2 &compclass section

This section specifes classes of generetors used for computation of type of curves. There are two different classes of generators: integrators/iterator used to compute orbits and the continuer used to compute bifurcation curves. Lines have the following form:

point_label number

CONTENT uses *number* to distingish between different methods of integration and continuation. *number* is **1** for integrators and iterator and **2** for continuers.

3.2.3 &menu section

This section decribes the structure of the **Type** menu in the Main window. The menu consists of two submenus. First submenu allows one to choose the type of initial point. Second submenu is used to choose the type of a curve computed from the initial point.

A submenu name is given by a line in the following form:

menu_name{

The end of submenu is marked by the line:

}

Other lines describe menu items. They may be in one of the following form:

point_label|*command_name*

specifies the type and gives the name for the menu command;

point_label

specifies the type (the name of the type is used form the menu command);

-

creates a separator line in the menu.

It is possible to group types in the submenu using the same technique.

Example. The following lines describe a submenu:

```
Initial point{
 O|Point
 -
 EP
}
```

Here **Initial point** is the name of submenu with two commands that choose **O** and **EP** point types respectively. The name of first command is given explicitly (**Point**); the name of second command is taken from the definition of the **EP** type and is the type's name.

3.2.4 &deftypes section

This sections specifies default types of curves set by CONTENT when the user selects a point on a computed curve as the inital point. This is done through the **Select|Inital point** menu or by clicking a special point in result graphic windows. Usually it is possible to compute curves of more than one type starting from a point of a given type. For example, CONTENT supports computation of orbits, equilibria and limit cycles started from an ordinary point of type **O**.

Each line has the following form:

curve_label point_label newcurve_label

where *point_label* is the type of the initial point, *curve_label* is the type of a curve the point belongs to, and *newcurve_label* is default type of new curve. Of course, the user may change curve type before computation.

Example. The line

```
LP ZH H
```

tells CONTENT to set the curve type to **H** when the user chooses a special point of the type **ZH** detected on a curve of the type **LP**.

3.2.5 &stdlinalg section

This section lists functions from a standard linear algebra package used by starters and generators. Each class of dynamical systems may have its own standard linear algebra package.

Format of lines in this section is described in 1.3.5. This section usually contains one line that include the **stdalg** file:

```
&&$(Paths.KERNEL)stdalg$(Extensions.DESC)
```

The kernel passes a table of pointers to functions from the package to starters and generators (chapter 7).

3.2.6 &utilities section

This section lists functions from the utility package used by starters and generators. Lines have the same form as in the **&stdlinalg** section. The functions are always available for starters and generators via a table of pointers passed by the kernel (chapter 7).

3.2.7 &functions section

This sections describes the interface between analysis procedures and functions specifying dynamical systems of the given class. The kernel uses this information to create C functions from the specification of dynamical system given by the user. Interface described in this section must be the same as used by the analysis procedurs.

The lines are grouped into blocks. Each block describes interface with one of functions specified by the user. The number of blocks depends of the class of dynamical systems. The following table gives the number of blocks (i.e. functions needed to specify dynamical systems) for each currently supported classes.

Class	Blocks	Comment
Maps	1	RHS
ODEs	1	RHS
PDEs	3	RHS and boundary conditions
DAEs	2	RHS and the matrix

The block describing RHS must be the first.

Each block in turn is divided into six subblocks. The first subblock describes a function, e.g. RHS. Other five subblocks describe derivatives of orders 1–5 of the function. There must be exactly five subblocks even if derivatives of some orders are not used by analysis procedures.

The structure of each subblock is shown below. At the right number of lines in each part are shown. ***** indicates variable number of lines.

menu item	1
prompt line	1
function name	1
header of function	1
defines **/***	*
tail of function	1
*undef*s **/***	*
dependent variables	1
independent variables **/***	*

In the following description we use examples taken from the ODEs class file **ode_g.con**. For illustration purpose we assume that only two classes labelled **Phase** and **Par** are defined with names **{x,y[3],z}** and **{a,b}**, respectively. ˜ is interpreted as end-of-line.

The *menu item* line specifies the name of the menu command in the **Specify** menu in the system specification dialog box. The command is used to position the cursor to the appropriate input field. Submenus are allowed. The line may have one of the following form:

submenu name{*menu command*
menu command
menu command}

where *submenu name* and *menu command* may be any text. This line in the first block is ignored — the input field for RHS is always available.

The *prompt line* specifies the text used as a label of the input field. It may be any text.

The *function name* line specifies the name of the function described by this block. It must be a legal C-function name and must be the same as in the *header of function*. It is recommended to use an underscore character, e.g. **_Rhs**. Names for derivative functions may be created by attaching suffixes to the name of the function: **_Rhs_Der1** etc.

The *header of function* line specifies the function header, that is its type, name, attributes, parameter list, and opening {. The name must be the same as in the *function name* line. For derivatives it may also declare arrays used to store the values of derivatives. Everything the user types in the input field becomes the body of the function (it is put after *defines*, see below). For example, the line for RHS might look like this (note it is splitted into several lines for illustration purpose only)

```
int $(Compiler.RHSATTR) _Rhs(double *_x,
                             double *_p,
                             double *_t,
                             double *_f) {˜
```

The line for first order derivatives of RHS might be

```
typedef double _d1T[@(%Phase)][@(%Pahse+%Par)];˜
static _d1T _d1;˜
double * $(Compiler.RHSATTR) _Rhs_Der1(double *_x,
                                       double *_p,
                                       double *_t) {˜
```

The *defines* is a series of lines. Lines are processed as described in 1.3.3 and then put after the header of the function. This allows to have **define** directives that map the names used by the user in system specification to names of formal parameters. **define** directives for RHS

```
@Phase {#define $Phase (*Phase(_x+&Phase))˜}
@Par   {#define $Par (*Par(_p+&Par))˜}
@Phase {#define $Phase_ (*Phase(_f+&Phase))˜}
```

are expanded to

```
#define x (*(_x+0))
#define y ((_x+1))
#define z (*(_x+4))
#define a (*(_p+0))
#define b (*(_p+1))
#define x_ (*(_f+0))
#define y_ ((_f+1))
#define z_ (*(_f+4))
```

(Note CONTENT replaces all ' characters in user input by underscores, e.g. `x'=x+a` becomes `x_=x+a`). Similarly, directives for first order derivatives

```
@Phase {#define $Phase (*Phase(_x+&Phase))~}
@Par   {#define $Par (*Par(_p+&Par))~}
@Phase {#define $Phase_ (*Phase*(_d1T*)(_d1+&Phase))~}
```

are expanded to

```
#define x (*(_x+0))
#define y ((_x+1))
#define z (*(_x+4))
#define a (*(_p+0))
#define b (*(_p+1))
#define x_ (**(_d1T*)(_d1+0))
#define y_ (*(_d1T*)(_d1+1))
#define z_ (**(_d1T*)(_d1+4))
```

Lines with comment that begin with two equal signs (i.e. `/*==...*/`) are put into intermediate C++ program only. This feature is used to define macros needed in the C++ program. To understand better how it all works it is worth to specify a differential-algebraic system with symbolical calculations of derivatives and then take a look at files **tmp_der.cc**, **tmp_der.c** and **dae_g.con**.

The *tail of function* specifies lines that terminate the function: a return statement and closing **}**. The lines put after the text typed by the user for the function. For RHS the line might be

```
return 0;~}~
```

For derivatives the function returns a poiniter to array

```
return _d1[0];~}~
```

The *undef* s is a series of lines. They are processed in the same way as *define*s and then put after the function tail. This allows to **undef** all names defined by *define*s.

The *dependent variables* line lists the names of dependent variables needed for building of symbolical derivatives of the function. It is usually the names to which assignments are made in the specification given by the user. This line is ignored for derivatives. For RHS of ODEs the line looks like

```
@Phase { $Phase_}
```

For differential-algebraic equations things are more complicated, see the file **dae_g.con**.

The *independent variables* is a series of lines. It lists the names of independent variables in respect to which the function need to be differentiated. Usually these are names of phase (state) variables and parameters

```
@Phase { $Phase}
@Par   { $Par}
```

This lines are ignored for derivatives.

3.2.8 &derivatives section

This section contains one line with a number that specifies the maximum order of derivatives needed for computations of curves of any supported type.

3.2.9 &visiblenames section

This section describes names of variables that user should give as a part of dynamical system specification. It also specifies the rules used to define names of other variables. The values of these variables are computed by analysis procedures and may be stored to archive or seen in the result windows.

Data from the section are used by the interface manager (1.1.1). The key notion is <u>class of visible names</u>. It is defined as a list of names (simple or one-dimensional arrays) either given by the user or predefined by the developer. Here are several examples of classes of visible names:

- Names for state variables and parameters of dynamical system are specified by the user,
- Names of eigenvalues are pre-defined by the developer (note that this requires the knowledge of the dimension of the system),
- Names of test functions (1.2.3) for each supported type of curves are pre-defined by the developer; for each type of curves the number and order of names depend on the analysis procedure which checks the values of test functions.

A class of visible names may be either basic or derived. Basic classes are those for which the user gives the names. For derived classes the developer assigns the names or gives the rules for generation of the names based on names from basic classes.

Each line in the **&visiblenames** section defines one class of visible names and has the following format:

code label v-name ~ definition ~ condition

*code*s, *label*s and *v-name*s are defined in1.3.3. Basic classes are listed first and separated from derived classes by the line with zero code. The *label* may be preceded by an asterisk character, e.g. ***Phase**. This means that curves of <u>all</u> types contain multiply points from this visible class. An example is the class of partial differential equations. A hyphen appended to the *label* instructs CONTENT not to write points from this class to archive by default. This setting may be changed through the archive filter dialog box opened by the **Options|Archive filter** command.

For each basic class there is an input field in the System specification dialog box. The user types here the names of variables. The v-name of a class is used as a prompt for the field. The class of phase (state) variables must be defined first.

For basic classes *definition*s must be empty; CONTENT takes the names from user's input. *condition*s have the form *relation number* where *relation* is one of the following: >, >=, ==. The *condition* is used by CONTENT to check the numbers of names specified by the user: The number of names must be in the given relation with the *number*. For example, == **1** means that only one name is required, >= **0** means names are optional, and > **1** means that at least one name must be given.

For derived classes *condition* must be empty. The *definition* is a list of names of variables, e.g. names of test functions for equilibria are **Brahching**, **Fold** and **Hopf**. @- and @&- loops are allowed (1.3.4), for example for eigenvalues the *definition*

```
@Phase&␣{␣Re_&Phase}␣@Phase&␣{␣Im_&Phase}
```

is expanded to the list of names

```
␣Re_0␣Re_1␣...␣Im_0␣Im_1␣...
```

The class labeled **UserFunc** is a special class for values of the user functions specified through the user functions dialog box (opened by the **Select|User functions** command).

Classes with codes that are less than 250 are called local. That means that different class files may define different codes for logically identical classes. For example, the class of standard functions, **StdFunc**, for ODES is coded by **13** whereas for PDEs the code is **11**.

Classes with codes that are greater than 250 are called global. Each global class must have the same code throughout all class files. The following global classes are defined in class files.

Code	Label	Description
252	SPECCL_MSUP	superposition for iterated maps
253	SPECCL_DIM	number of mathematical points in each point produced by a generator
254	SPECCL_AUS	choices made for user functions (ignore/detect/monitor/append)

The differences between local and global classes is that values from local classes may be components of the points computed by generators. Or they may be computed as a supplemental to points. Values from global classes usually characterize some non-computational properties of points and therefore must be treated differently.

3.2.10 &help section

This section specifies the name of a help file that contains all help topics related to the class of dynamical systems. Two topics whose keys are listed in the **&hierarchy** section of the main class file (3.1) must be present in the file.

If several file names are given they should be separated by spaces.

3.2.11 &windows section

This section describes classes of result windows specific for the given class of dynamical systems. Lines in this section have the same form as in the Windows file (2.3).

An example of specific result windows are staircase windows available for iterated maps only (see the **map_g.con** file).

Chapter 4

Curve type files

Curve type files describe analysis procedures that compute and process curves. There is one curve type file for each supported type of curves. When both types of initial point and curve are set through the **Type** menu the kernel reads appropriate line in the **&graph** section in the specific class file (3.2.1). It uses the *curve_type_file* from this line to get all information needed to create an environment for analysis procedures and load them into memory.

For each supported type of curves there are following C functions that compute, analyze and process points (1.2):

- the starter,
- the defining functions,
- the test functions,
- the processing functions,
- the adapter,
- the generator,
- the decoder.

Usually all but the generator functions are in the same dynamical object. The generator is separated from other analysis procedures because many different types of curves are computed by the same generator. For example, all bifurcation curves are computed by the continuer.

All curve type files have the &-format.

4.1 &call sections

The **&call** sections specify starters and generators. There is one **&call** section for each type of points from which curves of the given type may be computed (remind that each curve type file describe one type of curves, e.g. equilibria, Hopf, etc.). Let the labels of possible types of initial points are *LIP1*, *LIP2*, ... Then names of sections are **&call***LIP1*, **&call***LIP1*, ...

Each section contains two lines. The first line refers to the starter and second line refers to the generator.

Example. Consider the **__lc.con** file that describes the **limit cycle** type of curves. There is the following section in it

```
&callO
  __lc(p_lc_Starter)
  _conti(CurveCont)
```

This section describes the starter and the generator called to compute limit cycle curves from the points of type **O**. The starter is in the dynamic object **__lc** and has entry point **p_lc_Starter**. The generator's entry point is **CurveCont** and it is in the **_conti** dynamical object.

4.2 &types section

This section sets correspondence between test functions and detected singularities. It is used by generators to detect singularities based on the values of test functions. The correspondence is given by a matrix. Let *LC* is the label of the type described by the given curve type file, *LS1*, *LS2*, ..., *LSs* are labels of singularities (i.e. labels of types of special points) that may be detected on curves of type *LC*. Let f is the number of test functions (it must be equal to the number of lines in the **&testfuncs** section, 4.8). Then lines in the **&types** section are as follows

$$\begin{array}{ccccc}
\mathit{LS1} & v_{11} & v_{12} & \dots & v_{1f} \\
\mathit{LS2} & v_{21} & v_{22} & \dots & v_{2f} \\
\dots & \dots & \dots & \dots & \dots \\
\mathit{LSs} & v_{s1} & v_{s2} & \dots & v_{sf} \\
\mathit{LC} & \textbf{-} & \textbf{-} & \dots & \textbf{-}
\end{array}$$

Lines in this matrix describe the values of test functions that used to detect the singularities. Note that the last line represents the curve type and contains only minus signs. v_{ij} may be one of the following characters

Character	The value of test function
0	must be zero
1	must be non-zero
-	is irrelevant

If there is no singularities detectable for the given type of curves then there must be the only line with the label of the type. The orbits are examples of this kind (see the **__o.con** file).

Example. The **&types** section of the **__lc.con** file contains the following lines

```
BPC   0 - - -
LPC   1 0 - -
PD    - - 0 -
NS    1 1 - 0
LC    - - - -
```

According to this matrix points of four types may be detected on **LC** curves. These types have labels **BPC**, **LPC**, **PD**, and **NS**. They are defined in the **&graph** section of the **ode_g.con** file. Points on **LC** curves are detected as special points of the **NS** type if values of two first test functions are non-zeros and the value of the forth test function is zero.

4.3 &names section

This section lists labels of classes of visible names. The labels are defined in the **&visiblenames** section of the specific class file (3.2.9). Only names from these classes are visible to the user: the interface manager (1.1.1) puts the names into lists from which the user chooses values to be seen in result windows and written to the archive. The order is significant: some arrays passed to starters and generators are filled in accordance with this order (7.3.2).

The values that constitute points computed by analysis procedures are mapped to the names by appropriate decoder (4.11).

Each line in this section contains the *label* of one class of visible names. The *label* may be preceded by an asterisk character, e.g. ***Phase**. This means that curves of this type contain multiply points from this visible class. An example is limit cycle bifurcation curves each point on which codes one cycle. A hyphen appended to the *label* instructs CONTENT not to write points from this class to archive by default. This setting may be changed through the archive filter dialog box opened by the **Options|Archive filter** command.

Example. The **&names** section of the **__lc.con** file lists the following names

```
*Time
*Phase
Par
Period
Mult-
TestLc-
UserFunc-
StdFunc-
```

4.4 &data sections

The **&data** sections specify parameter files for starters and generators. Parameter files (chapter 5) are used by the interface manager to create windows through which the user sets and changes parameters of starters and generators such as coordinates of initial points, tolerances, etc. They are used also to allocate and fill memory for the values of the parameters. Pointers to these data areas are passed to starters and generators by the kernel (chapter 7).

There is one **&data** section for each type of points from which curves of the given type may be computed (that is there is one-to-one correspondence between **&data** and **&call** sections). If the labels of possible types of initial points are *LIP1*, *LIP2*, ..., then names of sections are **&data***LIP1*, **&data***LIP1*, ...

Each section contains two lines. They specify names of parameter files for the starter and the generator, respectively. The name of a file may follow by an identifier. In such a case the identifier is considered as 'defined' during processing the parameter file by the interface manager. This is explained in more details in chapter 5 and 5.3.

Example. Consider the **__lc.con** file that describes the **limit cycle** type of curves. There is the following section in it

```
&data0
  __lc.h p !starter
  _conti.h lc !generator
```

__lc.h is the name of parameter file of the starter that computes the first **LC** point from given **O** point. The name **p** is defined during processing the file. Similarly, **_conti.h** is the name of parameter file of the generator that computes **LC** curves. The name **lc** is defined during processing the file.

4.5 &stafuncs section

This section lists the functions from linear algebra packages used by the starter. Each line specify one function as described in 1.3.5.

The kernel builds a table of pointers to functions and passes it to the starter (chapter 7). The starter must declare the structure of the table preserving the number, order and types of pointers. There is the header file for each linear algebra package (currently **_linalg.h**, **_bvpalg.h** and **_rdalg.h**) that declares a structure with pointers to all functions from the package. Appropriate header file may be included if the list in this section has one-to-one correspondence with the fields of the structure defined in the file. Otherwise the structure must contain only the pointers to listed functions. Functions from the standard linear algebra package listed the the **&stdlinalg** section of the specific class file are always accessible through a special pointer passes by the kernel (described in 7.3.2, the **FuncParams** field).

4.6 &genfuncs section

This section lists the functions from linear algebra packages used by the generator. Each line specify one function as described in 1.3.5.

The kernel builds a table of pointers to functions and passes it to the generator (chapter 7). The generator must declare the structure of the table preserving the number, order and types of pointers. There is the header file for each linear algebra package (currently **_linalg.h**, **_bvpalg.h** and **_rdalg.h**) that declares a structure with pointers to all functions from the package. Appropriate header file may be included if the list in this section has one-to-one correspondence with the fields of the structure defined in the file. Otherwise the structure must contain only the pointers to listed functions. Functions from the standard linear algebra package listed the the **&stdlinalg** section of the specific class file are always accessible through a special pointer passes by the kernel.

4.7 &deffuncs section

This section specifies the defining functions (1.2.2) and the function that computes the Jacobian of the defining functions. They are used by the generator.

The section contains two lines that have the form described in 1.3.5.

4.8 &testfuncs section

This section specifies the test functions (1.2.3) that may be monitored along curves of the given type.

Each line contains one function and has the form described in 1.3.5. The number of lines must be the same as in the **&types** section (4.2).

4.9 &procfuncs section

This section specifies the processing functions (1.2.4). Each line has the form described in 1.3.5.

The number of lines must be one more than the number of singularities. The singularities are listed in the **&types** section (4.2); the number is denoted by s in the description.

The processing functions from second one correspond to singularities. The generator calls a processing function when it detects corresponding singularity. The function usually calculates some numerical characteristics of the singularity and prepares appropriate message to be shown in the main window.

The first processing function is called <u>default processor</u>. The default processor does not related to singularities. It is called by the generator for each computed point, not only for special points. The default processor computes the values of standard functions usually defined in the class labeled **StdFunc**. It may also do some auxiliary computations.

4.10 &adapters section

This section specifies the adapter (1.2.6) and must contain one line in the form described in 1.3.5.

The adapter may do nothing but return immediately.

4.11 &trans section

This section specifies the decoder (1.2.7) and must contain one line in the form described in 1.3.5.

The decoder maps the values from points computed by the generators to values from the classes of visible names listed in the **&names** section (4.3). The kernel-decoder interface is described in chapter 7.

4.12 &slave section

This section is optional. If present the section specifies a secondary set of analysis procedures, i.e. the starter, the generator, etc. Secondary starter and generator may be called by the primary starter. An example of this kind is computation of **LC** curves (limit cycles) from **O** points (ordinary points). The first step taken by the starter is to build an orbit as an approximation to the first limit cycle. This is done by calling one of the integrators.

Slave starter and generator may have parameter windows but are not allowed to write to the archive. The primary starter is responsible for setting all data necessary to call them and interception all output from them. From the slave starter and generator point of view they run in a normal way, i.e. as if they are primary starter and generator.

The name of the section is **&slave***LIP* where *LIP* is the *label* of a type of initial point. There is only one line in the section. It contains the *label* of type of curves. If the type is LC then analysis procedures for computation of curves of type LC from points of type *LIP* are made available for the primary starter.

Chapter 7 describes interface between primary and secondary starters.

During processing parameter files of slave starter and generator the **_slv** name is defined (4.4, 5.3).

4.13 Different implementations of analysis procedures

CONTENT supports the ability to have different <u>methods</u> of computation of the same mathematical object. Orbits of ODEs are an example of this kind. Analysis procedures for each supported method are independent from each other. If several methods of defining or computing curves are available the kernel creates the **Method** menu in the starter or generator window. The menu allows the user to pick up desired method.

Curve type files for multi-method curves have basically the same format. Corresponding lines related to different methods are concatenated into one line and their contents are separated by bars. Lines with the same information need not be merged.

For example, the **&callO** section of the **__o.con** file contains the following lines (names of functions were changed to shorten the line)

```
__o(o_Starter)
_integr1(rk_Orbit) | _integr(eu_Orbit) | _integr2(r5_Orbit)
```

It describes three method of computing orbits and specifies that for all these methods the same starter should be used.

The **&callH** section of the **__h.con** file illustrates the description of different methods of defining the Hopf curves computed by the same generator

```
__h(h_Starter) | __h1(h_h1_Starter)  | __h2(h_h2_Starter)
_conti(CurveCont)
```

There are some additional sections for the multi-method types.

4.13.1 &method section

This section must be the first in the file. It specifies the names of methods and commands CONTENT put into the **Method** menus. The commands are used to choose methods.

The section contains two lines, for different definitions of the type and different ways of computations, respectively. The names are separated by bars. If there is only one method then the corresponding line must contain only a hyphen.

Examples. The **&method** section of the **__o.con** file names three methods of computation of curves and specifies that there is the only method of definition of the **O** type (i.e. there is only one starter used to compute orbits)

```
-
Runge-Kutta  | Euler | RADAU5
```

The **&method** section of the **__h.con** file names three methods of definition of Hopf curves and specifies that all such curves are computed by the same generator.

```
standard | bordered biproduct | bordered square
-
```

4.13.2 &help sections

Help topics associated with different methods are different. The keys of the topics (chapter 6) specific for each method are provided in the **&help** sections.

For each type of initial point *LIP* there is the **&help***LIP* section with two lines in it. The first line has the following form

init_point_hk curve_hk starter_hk | *init_point_hk curve_hk starter_hk* ...

and lists help keys for topics describing types of initial points, curves and starter's window for different methods of defining the given type, respectively.

The second line specifies help keys for topics describing generator window for different methods of computation.

If there is the only method corresponding line must contain only a hyphen.

Keys listed in these sections have precedence over keys from the **&graph** section of the corresponding class file.

Example. The **&HelpO** section of the **__o.con** file lists the help keys of topics describing generator window for each of three supported methods of integration

```
-
method_RK | method_EU | method_RD5
```

Chapter 5

Parameter files

Parameter files describe parameters of starters and generators. They are used by CONTENT in several ways.

1. Each starter and generator includes (via the **#include** directive) its own parameter file to declare a structure that contains parameters.
2. The manager of dynamic objects (1.1.3) allocates and fills memory for parameters and passes pointers to them to starters and generators. This allows starters and generators to access the values of parameters.
3. The interface manager (1.1.1) creates windows that the user uses to change the values of parameters before computation of curves.
4. The archive manager (1.1.2) writes to the archive the values of parameters used to compute curves. This allows the user to recompute and export curves.

5.1 Structure of files

The parameter files are ASCII files. They contain declarations of structures that describe parameters of starters and generators. The language is **C**.

Each parameter file is a part of source text of appropriate starter or generator. It may contain comments, one-line macro definitions and must contain one declaration of a structure in the following form

typedef struct *struct_name* **{**
. . . .
} *type_name*, ******type_ptr_name*;

The members of the structure may be of the following types: **char**, **int**, **double**, **char***, **int***, **double***, and **FuncPar** (described in 5.4).

Information provided by such declaration is not sufficient to visualize the members of the structure. This insufficiency is compensated by additional information from so called **$**-comments that follow each member declaration. A comment is called **$**-comment if its first character is a **$** character:

/*$ . . . . . */

$-comments are further classified according the second letter:

/*$t...*/ are title comments.

/*$=...*/** and **/*$!...*/ are conditional comments.

/*$␣...*/ are unconditional comments.

The title comments are used for visualization purpose only. Each but the first **$t**-comment instructs the interface manager to create a title line in the parameter window using the text from the comment; the first **$t**-comment in the file gives the window's title. The conditional comments are basically the same as unconditional comments. The differenes are outlined in 5.3. The unconditional **$**-comments are described in 5.2.

Therefore, each member of the structure is declares as follows:

type name; *ordinary_comment $-comments*

The *type* is one of the types listed above. The *name* is the name of the member; it is used in the source code of the starter or generator to refer to the particular value. The *ordinary_comment* is optional and may contain any text. The *$-comments* is a series of **$**-comments each of which but the last one is a conditional comment; the last comment must be an unconditional comment.

Note that **$t**-comments are not related to members of the structure and may be placed before them where necessary. They are used to group parameters visually.

5.2 Data in $-comments

Each unconditional **$**-comment has the following form:

/*$␣*dim*␣|␣*v-indicator*␣*r-indicator*␣*external_names*␣|␣*init_list*␣*class_code*␣***/**

The following sections describe each component in details.

5.2.1 Dimension

The *dim* is a number or an expression that evaluates a number. Special symbols **%***class* denoted dimensions of classes of visible names are allowed (1.3.4). For example, the **%Par** expression gives the number of names in the **Par** class.

The value specifies the number of components for the member. It must be **1** for non-pointers. For each pointer the kernel allocates requested amount of memory and put its address into the pointer. The kernel also frees this memory when it is no longer needed.

Some values have special meaning.

-1 Instructs the kernel not to allocate and free memory. It also does not store the contents of the memory into the archive. It is assumed that analysis procedures are fully responsible for the management of such pointers. The type of the member may be **void***. An example of this kind is the pointer to coordinates of initial point in a generator data area. Appropriate starters allocate and fill memory, set the pointer and free the memory after computations.

-2 Specifies that the amount of memory in bytes should be taken from the next member of the structure (which must be of type **int**). An example of this kind is the pointer to coordinates of the initial point for limit cycle bifurcation curves. The initial point here is a series of points in the phase space. The total amount of memory needed may be calculated only by appropriate starter.

-10-*d* The same as **-1** but specifies for the starter or generator that the real number of components is d. An example of this kind are members used by functional parameters (see the **__lc.h**) file.

5.2.2 Visibility indicator

The *v-indicator*, called the visibility indicator, specifies whether this member is visible in the window or not. The values of the indicator may be + or - for visible and invisible members, respectively. For example, invisible state may be used in conjunction with conditional comments to make visibility of the member depending on the type of initial point.

5.2.3 Representation indicator

The *r-indicator*, called the representation indicator, specifies the form in which the values of the member are represented in the window. It is one letter from the following list.

n The values are represented as numbers using the **%.***p***g** format. p is the value of the **PRECISION** key from the [**Params**] section of the settings file.

N The same as **n**. In addition the user can highlight external names (5.2.4) of the member by clicking them by mouse. The next member must be an invisible pointer to an array of **char** of the same dimension. The array is used to keep the highlight state of the names. This option is used by starters to distinguish active and frozen parameters of dynamical systems.

o The same as **n** with the exception that the user cannot change the values (output only field).

s The possible values are represented by the list of external names (5.2.4). The member must have type **int** or **int***. The user chooses desired value (which is an integer from **0** to number of names minus one) by clicking appropriate name.

S The same as **s** with the exception that only one current name is visible. To set another value one has to click several times until the desired name appears.

f The member is a functional parameter (5.4).

For pointers the same from is used for all elements.

5.2.4 External names

The *external_names* is a list of space separated names the interface manager uses to label the values of the member in the window. The external names has nothing in common with the name of the member.

For an invisible member the the *external_names* may be omitted. For a visible member the number of external names must by the same as its dimension. @-loops (1.3.4) may be used to generate names from classes of visible names.

For example, the pointer to the values of parameters of a dynamical system may be declared as follows

```
double *P;    /* Init point: parameters */
              /*$ %Par | + N @Par { $Par} | "0" %LCode(Par) */
char *Active; /* Highlight state of parameters */
              /*$ %Par | - n _ActivePar | "0" */
```

An external name may follow by an expression enclosed in parentheses, e.g. **Zero-Hopf(%Phase-3)**. In such a case the value is invisible only if the expression has non-negative value. In the above example the value is visible for dynamical systems of dimensions higher than 2.

External names may be used by the user in functional parameters (5.4).

5.2.5 Default values

The *init_list* is a comma separated list of numbers that are used as default values for the member. The list must be enclosed in double quotes, e.g. **"0,1,2"**. The last or only value is used if the list has less numbers than the dimension of the member.

5.2.6 Code of class of visible names

The *class_code* is the code of class of visible names that corresponds to the member. The code is optional and should only be given for those members of the structure that correspond to basic classes (1.3.4). Examples are members that hold values for state (phase) variables, parameters of dynamical systems, time variable.

The code is used by the archive manager (1.1.2). When the user selects a point as the initial point (e.g. by 'taking' a special point on a curve) the manager need to assign coordinates of the point to members of the structure to allow the user to see and change the coordinates before computation. The code provides the link between coordinates of a point and the member.

The *class_code* may be given as a number assigned to the class in the **&visiblenames** section of the class file (3.2.9). However, the preferred way is to use class labels. The interface manager supports two special functions, **%LCode** and **%Code**.

If *code* and *label* are the code and the label of a class of visible names, respectively, then **%LCode(***label***)**=*code*. CONTENT searches the *label* in the **&names** section of appropriate curve type file (4.3) and then looks at the **&visiblenames** section of the class file to find the *code*. The **%LCode** function should be applied to local classes only.

For the **%Code** function the search is done in the **&visiblenames** section of the class file. The function should be applied to global classes only.

5.3 Conditional $-comments

Starters computed first points for the same type of curves from initial points of different types (e.g. the **EP** curves from **BP** and **EP** points) often use the same parameter file. In such situations some members of the structure should have different visualization attributes. For example, the **primary/secondary** switch must be visible if the type of initial point is **BP** and should be invisible otherwise. Conditional **$**-comments with the mechanism of 'defined' identifiers described in 4.4 provide this feature.

Conditional **$**-comments contains the same data as unconditional comments (5.2) with the extension that they have a condition tested by the interface manager. The condition immediately follows the $ sign and has one of the two forms:

/*$=*id* . . . */

or

/*$!*id* . . . */

When the interface manager reads the comment is performs the following checks. If the condition is =*id* and the *id* is undefined or the condition is **!***id* and the *id* is defined then the comment is ignored and the manager reads the next **$**-comment. Otherwise, the manager processes the comment and then skips all successive **$**-comments until the next member.

As an example let us consider the following member declared in the **_h.h** file

```
int *ZeroTestFunc; /* 0 - NOT zeroed; 1 - zeroed */
                   /*$=bt 4 | - n | "0,1,1,0"    */
                   /*$=zh 4 | - n | "0,0,1,0"    */
                   /*$    4 | - n | "0"          */
```

If **bt** is defined then only the first **$**-comment is processed, Similary, if **zh** is defined then only the second comment is processed. If neither **bt** nor **zh** is defined then only the last comment is processed.

5.4 Functional parameters

Functional parameters of starters and generators are functions supplied by the user. To specify such a parameter the user should click the word **Edit...** that follows the name of the parameter in appropriate starter or generator window. CONTENT opens a window where the user types in C statements in which all names from the window may be used. Generally, starters and generators may use functional parameters for various purposes. Currently many starters have such parameters that they call just before locating the first point. This allows the user, for example, to assign coordinates of inital point by providing a function instead of doing this 'manually'. Let us consider two examples.

Example 1. For dynamical system $x' = x*x + p*p - 1$ the following function might be given for the **EP->EP** starter:

```
x=sqrt(1-p*p);
ActivePar(p);
```

The first line assigns the initial value to the phase variable. **ActivePar** is a macro defined in the starter's parameter file. It is used to set an 'active' state of parameters. i.e. to allow they to vary.

Example 2. For computation of limit cycle bifurcation curves an 'initial cycle' may be needed. The functional parameter of the **LC->LC** starter provides the ability to supply a program that calculates coordeintas or reads them from a file. Here is how it might look for a system with phase variables **x**, **y**, and **z** and time variable **t**:

```
static FILE * f;
switch (ip) {
  case -1:
    f=fopen("tor.dat","r");
    return f ? 0:1;
  case -2:
    fclose(f);
    return 0;
  default:
    if (fscanf(f," %lg %lg %lg %lg",&t,&x,&y,&z)==4) {
      t*=Period;
      return 0; /* Point has been read */
    } else {
      return 1; /* No more points */
    }
}
```

ip is a parameter passed to the functional parameter by the starter. **-1** means 'initalize', **-2** means 'terminate'. The starter calls the function until it returns non-zero value indicating that all points have been generated.

Note again, that the number of funcional parameters, their semantic as well as the interface to them is up to the starters and generators.

Functional parameters must be declared by the **FuncPar** macro defined in the **stagen1.h** file (which is included by the **stagen.h** file). The generic form of declarations is

FuncPar(*type*,*func_name*,(*arg_list*)**);**

where *type* is the type of return value, *func_name* is the name of the C function CONTENT creats for this parameter, *arg_list* is the list of arguments. One of the arguments must be

struct *struct_name* ***_ptr**

where *struct_name* is the name of the structure (5.1). This pointer provides the function with access to the members of the structure.

Here are declarations of functional parameters described in the above examples:

```
FuncPar(void,SetInitPoint,(struct ___ep_Data * _ptr));
                  /*$=bp 1 | - f SetInitPoint | "" */
                  /*$    1 | + f SetInitPoint | "" */
```

and

```
FuncPar(int,SetInitPoint,(int ip, struct ___lc_Data * _ptr));
                  /*$=p  1 | + f SetInitPoint | "" */
                  /*$=h  1 | + f SetInitPoint | "" */
                  /*$=lc 1 | + f SetInitCycle | "" */
                  /*$    1 | - f              | "" */
```

Note that the name of a C function may be different from the external (i.e. visible in the parameter window) name.

The **stagen1.h** file contains two macros that facilitate calling of functional parameters by starters and generators. Let us assume for the following description that *DataPtr* is a pointer to the data area of a starter or a generator (the value can be get from the **StaPtr** or **GenPtr** fields of the structure passed to starters and generators; see 7.3.2) with one functional parameter, *SetInitPoint*, declared like this:

```
  typedef struct __Data {
    . . . . . . . . . . . .
    FuncPar(void,SetInitPoint,(struct __Data *_ptr));
    . . . . . . . . . . . .
  } _Data, *_DataPtr;
  . . . . . . . . . . . . .
  _DataPtr DataPtr;
```

The **FuncParActive** macro is used to test the status of functional parameters. The convention is that any functional parameter is ignored (not called) until it is specified and activated . The user actives the parameter by clicking its name in appropriate parameter window. CONTENT marks active parameters by highlighting their names. Active parameter becomes inactvie when the user clicks its name again. The macro returns **TRUE** if the parameter is active and returns **FALSE** otherwise.

FuncParActive(*DataPtr*->*SetInitPoint***)**

The **FuncParCall** macro calls functional parameters:

FuncParCall(*DataPtr*->*SetInitPoint***)(***parameters***)**

Note that the number and types of *parameters* must be the same as in declaration of *SetInitPoint*.

Chapter 6

Help files

Help files contain texts used by on-line help system that is called by the user through appropriate menu commands and buttons during sessions.

For each source help file written by the developer there is associated target help file created by a special program called help compiler. The compiler is a part of the developer's toolkit. Target help files are used by CONTENT. The extension of any source file is **.hel**. Target files have the **.hlp** extension. The help compiler reads the **helplist.cpf** file each line of which must contain names of source and target files separated by blanks. File extensions must be present as well.

Several (target) help files are supplied with CONTENT :

File	Contains topics about
content	kernel
contw1ds	staircase result windows
contw2dg	2D result windows
contw3dg	3D result windows
contwnum	numeric result windows
contwtab	tabular result windows
contds	general terms
cont_pd1	PDEs
cont_ode	ODEs
cont_map	iterated maps
cont_dae	differential-algebraic systems

The developer may create new help files. The following table tells you where references to new help files must be put depending on their contents.

Contents	Put/Add filename to
new curve type	add to **&help** section of class file
new class of dynamical systems	put to **&help** section of class file
general	add to **Files** key in [**Help**] section of the settings file
new kind of windows	put to **windows.con** or **&windows** section of class file

The language used to write help topics is similar to LaTeXwith many limitations and some extensions. It is described in the following sections in details.

This chapter describes the structure of source help files. Any text between a **%** character and the following end of line in a source file is considered as a comment and ignored.

6.1 Topics

A help file consists of a series of topics. A topic is a minimal unit of text that on-line help system can display in the help window. Each topic describes one term, mathematical method, window, etc. Topics may refer to other topics through hyper-links (6.3).

Each topic is delimited by the **\topic** and **\end** commands:

\topic *key* **{***title***}**

topic_text

\end

The *key* is an identifier called the help key of the topic. It is used in **\jump** commands to create hyper-links referring the topic.

The *title* is the title of the topic. It may contain spaces and is used to represent the topic in the search dialog box that allows the user to browse and pick up particular topics. If *title* contains only one hyphen, e.g.

\topic *key* **{-}**

then the topic is not listed in the dialog box. It may be used in **\include** commands only (6.4).

The *topic_text* may contain any text and commands described below except **\topic** and **\end**.

6.2 Conditional directives

Different parts of the text may be selectively processed or ignored. Selection is governed by conditional directives similar to those found in the **C** programming language. Unlike in **C** conditional directives cannot be nested.

\define directives are used to 'define' identifiers:

\define *id*

makes the *id* 'defined' for the rest of the help file.

\ifdef directive checks whether given *id* is defined. If it is then *then_text* is processed and *else_text* is ignored. Otherwise, *then_text* is ignored and *else_text* is processed:

\ifdef *id*
$\boxed{\textit{then\_text}}$ %processed if *id* is defined
\else
$\boxed{\textit{else\_text}}$ %processed if *id* is not defined
\endif

\ifndef is similar to **\ifdef** but tests opposite condition:

\ifndef *id*
$\boxed{\textit{then\_text}}$ %processed if *id* is not defined
\else
$\boxed{\textit{else\_text}}$ %processed if *id* is defined
\endif

6.3 Intertopic references

A **\jump** command creates a hyper-link. The hyper-link is a part of the text visually marked by different color and underlined that reacts to mouse clicks by displaying another topic. It has the form:

\jump *key* **{***text***}**

key is the key of a topic to be shown. *text* is the text of the hyper-link replacing the command in the target file. When the user clicks somewhere at the *text* CONTENT displays the topic associated with the *key*. Note that the topic may by in a different help file. the only requirement is that the file must be already loaded by CONTENT.

On-line help system of CONTENT maintains several so called help-variables. Values of a help-variable are keys of help topics. The kernel changes the values of the variables during the sessions on the basis of choices made by the user and data from configuration files. Any help-variable may by used as the *key* in **\jump** commands. There are following help-variables.

***curclass** The key of the topic that describes currently chosen class of dynamical systems.

***curtypes** The key of the topic that lists all supported types of curves for the current class of dynamical system.

***curpoint** The key of the topic that describes currently set type of initial points.

***curcurve** The key of the topic that describes currently set type of curves.

***curstarter** The key of the topic that describes parameter window of currently loaded starter.

***curgenerator** The key of the topic that describes parameter window of currently loaded generator.

For example, if the **ODEs** class is chosen then the value of the ***curclass** variable is **ode_Class**. If the user chooses the **PDEs** class the kernel changes the value of the variable to **pde1_Class**. The keys are taken from the main class file **classes.con** (3.1).

6.4 Saving typing

A piece of text may appear in several places in a help file. To save typing put such a text into separate topic with the dummy title **{-}**. Then use **\include** commands to direct CONTENT to process the text from the topic.

The **\include** command has the following form:

\include *key*

where *key* is the key of a topic to be processed. *key* may be one of the help-variables (6.3).

6.5 Producing text 'as is'

The **\verbatim** command allows one to put lines into the target file exactly as they are without any processing. The command has the form:

\verbatim n

The given number n of lines immediately following the command are copied into the target file as they are. This command is useful for producing parts of programs in help files.

\verbatim commands cannot be nested.

6.6 Descriptions

The **\desc** command allows one to produce a list of terms alone with their explanations. The list usually looks like this:

Button	**Description**
Ok	Performs requested actions.
Cancel	Aborts and closes the window.
Help	Opens help window.
. . .	. . . . .

The **\desc** command should be used as follows:

\desc n

term$_1$ \> *explanation*$_1$ **\par**

term$_2$ \> *explanation*$_2$ **\par**

. . . .

term$_n$ \> *explanation*$_n$ **\par**

Here *term*$_i$ and *explanation*$_i$ may be any text containing formatting commands.

explanation$_i$ may occupy several lines and must be followed by the **\par** command.

Neither *term*$_i$ nor *explanation*$_i$ cannot contain another **\desc** command.

The source text for the above example is shown below:

```
\desc 5      % four pairs term-explanation
  {\bf Button} \> {\bf Description}           \par
  {\bf Ok    } \> Performs requested actions.  \par
  {\bf Cancel} \> Aborts and closes the window.\par
  {\bf Help  } \> Opens help window.           \par
  {\bf . . . } \> . . . . .                    \par
```

6.7 Scope of commands

Each command that does not produces output but sets the mode of processing has the scope associated with it. The scope is defined as a part of text where the mode set by the command is in effect. The scope always begins with the command itself. If the command is in a matching pair of braces then closing matching brace ends the scope. In the following example the scope of a command is underlined:

...{\bf ... {...} ...}...

At the end of the scope the previous mode is restored.

If the command is outside and matching pair of braces then the scope ends at the end of file.

6.8 Paragraphs

A paragraph is any text terminated by the **\par** command. The command has no parameters and may be placed everywhere you feel it needed. Text after the command begins from new line.

Several commands are used to specify different formatting characteristics of paragraphs.

\li *hdist*

Sets the left margin.

\fi *hdist*

Sets the indent of the first line of paragraphs. It is measured from the left margin and may be negative.

\sb *vdist*

Increases vertical spacing before paragraphs.

\sa *vdist*

Increases vertical spacing after paragraphs.

The unit of horizontal distances is the width of the most wide character from default font. The unit of vertical distances is the height of characters from default font. *hdist* and *vdist* may be integer or real numbers.

6.9 Keeping things together

Two commands are provided for 'gluing' tokens.

The \+ command produces so called <u>hard space</u> . Hard spaces have the following properties: 1) lines are never broken at hard spaces, and 2) each hard space becomes exactly one ordinary interword space in target file. Use this command to prevent lines breaking and increasing spaces between words where necessary.

The \| command is used to keep together characters that do not form a single token. No space is produced instead of the command. Consider an example. Suppose one wants to produce the following text without spaces: `<field>`. By default, the help compiler breaks the text into three tokens ($<$, **field**, and $>$) and inserts spaces between them. To prevent this the \| commands should be used: `<\|field\|>`.

6.10 Alignment of lines

The **\qc** command centers its argument between left and right margins:

\qc {*any_text***}**

The **\qr** command adjusts its argument to the right margin:

\qr {*any_text***}**

Note that *any_text* may contain mathematical formulas (6.12) and pictures (6.13).

6.11 Fonts and colors

The **\bf** command selects the bold font:

\bf

The **\f** command may be used to select one of the fonts used by help system:

\f *font*

where *font* is one of the following:

*	text font.
math	mathematical font.
mathl	large mathematical font.
matht	text font used in formulas.
scrt	text font used in sub- and superscripts.
scrm	mathematical font used in sub- and superscripts.
prog	font for listings.
b*	bold text font (the same as **\bf**).

The help system selects these fonts when required so you seldom need to select them by yourself.

The **\cf** command sets colors:

\cf *color*

where *color* is one of the following: **black**, **red**, **green**, **blue**, **cyan**, **magenta**, **yellow**, **gray**, **white**.

6.12 Mathematical formulas

Mathematical formulas may appear everywhere in help text. Formulas must be enclosed in matching pair of the **$** characters, i.e. **$***forlula***$**

Commands availabe only in formulas are described below. Apart them the following commands may be used: **f**, **bf** and **cf**.

6.12.1 Subscripts and superscripts

Superscripts and subscripts are made with the ^ and _ commands. If an expression has both of them the superscript must go first: *expr^super_sub* is correct while *expr_sub^super* is not.

You may enclose expressions, superscripts and subscripts in curly braces. If ^ or _ command does not follow by **{** then next number or identifier is considered as superscript or subscript.

6.12.2 Special characters

Many mathematical and special characters can be used in formulas. The following table lists the commands to produce them.

A	**\Alpha**	$\cdot$	**\cdot**	π	**\pi**
B	**\Beta**	χ	**\chi**	$\pm$	**\pm**
X	**\Chi**	©	**\copyright**	$\prime$	**\prime**
Δ	**\Delta**	δ	**\delta**	ψ	**\psi**
E	**\Epsilon**		**\empty**	$\rangle$	**\rangle**
H	**\Eta**	$\emptyset$	**\emptyset**	ρ	**\rho**
Γ	**\Gamma**	ϵ	**\epsilon**	$\rightarrow$	**\rightarrow**
I	**\Iota**	$\equiv$	**\equiv**	σ	**\sigma**
K	**\Kappa**	η	**\eta**	$\sim$	**\sim**
Λ	**\Lambda**	γ	**\gamma**	$\subset$	**\subset**
M	**\Mu**	$\geq$	**\geq**	$\supset$	**\supset**
N	**\Nu**	$\in$	**\in**	τ	**\tau**
Ω	**\Omega**	∞	**\infty**	θ	**\theta**
O	**\Omicron**	ι	**\iota**	$\times$	**\times**
Φ	**\Phi**	κ	**\kappa**	υ	**\upsilon**
Π	**\Pi**	λ	**\lambda**	ε	**\varepsilon**
Ψ	**\Psi**	$\langle$	**\langle**	φ	**\varphi**
P	**\Rho**	$\ldots$	**\ldots**	ϱ	**\varrho**
Σ	**\Sigma**	$\leq$	**\leq**	ς	**\varsigma**
T	**\Tau**	μ	**\mu**	ϑ	**\vartheta**
Θ	**\Theta**	∇	**\nabla**	$\vee$	**\vee**
Υ	**\Upsilon**	$\neq$	**\neq**	$\wedge$	**\wedge**
Ξ	**\Xi**	ν	**\nu**	ξ	**\xi**
Z	**\Zeta**	$\oplus$	**\oplus**	ζ	**\zeta**
α	**\alpha**	$\otimes$	**\otimes**	$\{$	**\{**
$\approx$	**\approx**	ω	**\omega**	$\}$	**\}**
$-$	**\arrowhorizex**	o	**\omicron**	$\_$	**\underscore**
β	**\beta**	∂	**\partial**		
$\bullet$	**\bullet**	ϕ	**\phi**		

6.12.3 Spacing in formulas

Spaces typed in formulas are ignored by the help compiler and are not copied into target file. However, you may instruct help system to add some horizontal space using one of the following command. They have no operands.

\,	thin space
\:	medium space
\;	thick space
\!	negative thin space

6.12.4 Accents

The accent commands produce various accents in formulas. In the table below the letter *a* denotes any text.

$\hat{a}$	**\hat**	$\tilde{a}$	**\tilde**	$\bar{a}$	**\bar**	$\dot{a}$	**\dot**	$\dot{\bar{a}}$	**\bardot**

6.12.5 Square roots

The **\sqrt** command produces the square root of its argument:

\sqrt{*expr***}**

For example, the command `$\sqrt{x^2+y^2}$` produced $\sqrt{x^2+y^2}$.

6.12.6 Fractions

The **\frac** command is used to produce fractions:

\frac{*numerator***}{***denominator***}**

For example, the command `$y=\frac{x}{x^2+1}$` produces $y = \frac{x}{x^2+1}$.

6.12.7 Sums, products, and integrals

Sums, products, and integrals are produced by the **\sum**, **\prod**, and **\int** commands, respectively. Large versions of these symbols are produced by the **\Sum**, **\Prod**, and **\Int** commands. Each command has three arguments: subscript, superscript, and the expression, in the order. They are shown below:

\sum{*subscr***}{***superscr***}{***expr***}**
\prod{*subscr***}{***superscr***}{***expr***}**
\int{*subscr***}{***superscr***}{***expr***}**

In the following examples both source text and produced formulas are shown:

`$\sum{i=0}{\infty}{\frac{1}{i}}=\infty$`	$\sum_{i=0}^{\infty} \frac{1}{i} = \infty$
`$\prod{i=1}{n}{i}=n!$`	$\prod_{i=1}^{n} i = n!$
`$\int{\empty}{\empty}{x^2 dx=\frac{x^3}{3}}$`	$\int x^2 dx = \frac{x^3}{3}$

6.12.8 Limits, minima, and such

Several mathematical symbols may require another expressions being put below them. These symbols are produced by the following commands:

\lim{*subscr***}{***expr***}**
\min{*subscr***}{***expr***}**
\max{*subscr***}{***expr***}**
\inf{*subscr***}{***expr***}**
\sup{*subscr***}{***expr***}**

In the following examples both source text and produced formulas are shown:

`$\lim{x\rightarrow 0}{\frac{sin x}{x}}=0$`	$\lim_{x\to 0} \frac{\sin x}{x} = 0$
`$\inf{x^2>2}{x}=\sqrt{2}$`	$\inf_{x^2>2} x = \sqrt{2}$

6.12.9 Delimiters

Delimiters of regular size are produced by commands shown below:

$\{$	\{	(	(	[	[	$\vert$	|
$\}$	\}	)	)	]	]		

Delimiters big enough to 'fit around' the expressions that they surround are produced by the **\brackets***lr* command:

\brackets*lr***{***expr***}**

where l and r specify delimiters produced on the left- and right-hand sides of the *expr*, respectively. l and r may be one of the following characters:

l	.	$\vert$	(	[	$\{$
r	.	$\vert$	)	]	$\}$

Dot means 'no delimiter' and may be used to produce a delimiter on one side of the expresions only.

In the following examples both source text and produced formulas are shown:

`$\brackets(){\frac{1}{2}+\frac{1}{4}}=0.75$`	$\left(\frac{1}{2}+\frac{1}{4}\right)=0.75$
`$\brackets||{-\frac{1}{2}}=\frac{1}{2}$`	$\left\vert-\frac{1}{2}\right\vert=\frac{1}{2}$
`$\brackets.|{x^2}_{x=3}=9$`	$\left.x^2\right\vert_{x=3}=9$

6.12.10 2-dimensional arrays

The help system supports four kind of 2-dimensional arrays: matrices, determinants, systems (of equations), and tables. They all are tables of items and differ from each other only by delimiters that enclose them:

Array	Delimiters
matrix	(*items*)
determinant	$\vert$*items*$\vert$
system	$\{$*items*
table	*items*

Commands for arrays are **\matrix**, **\determinant**, **\system**, and **\table**. They have the same arguments:

rows colalignment hmin wadd items

rows is the number of rows.

colalignment specifies the number of columns and the alignment of items in each column. The number of columns is equal to the number of the letters in the argument. Alignment of j-th column is given by the j-th letter which can be

- **l** flush items left
- **c** center items
- **r** flush items right

hmin is a real number expressed in the height of letters from the default text font. The number specifies the minimal height of all items. If it is zero then each row has its own natural height.

wadd is a real number expressed in the width of the most wide character in the default text font. The number specifies extra horizontal space added between adjacent columns.

items is a series of items each of which must be enclosed in curly braces: **{***item***}**. The number of items must by equal to *rows*×strlen(*colalignment*) and they must be listed row by row without delimiters.

Here are some examples. The system of equations

$$\left\{ \begin{array}{l} 2x + 3y = 4 \\ 5x + 6y = 7 \end{array} \right.$$

is produced by the following command:

```
$\system 2 cll 0 0 {2x+3y} {=} {4} {5x+6y} {=} {7}$
```

Array commands may be nested. The matrix

$$\begin{pmatrix} \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} & 0 & 0 \\ 0 & \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} & 0 \\ 0 & 0 & \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} \end{pmatrix}$$

is produced by the following command:

```
$\matrix 3 ccc 0 0
 {\matrix 2 cc 0 0 {a_{11}}{a_{12}}{a_{21}}{a_{22}}} {0} {0}
 {0} {\matrix 2 cc 0 0 {b_{11}}{b_{12}}{b_{21}}{b_{22}}} {0}
 {0} {0} {\matrix 2 cc 0 0 {c_{11}}{c_{12}}{c_{21}}{c_{22}}}
$
```

6.13 Pictures

The **\picture** command produces a picture. The command has two arguments:

\picture *ratio filename*

where *ratio* is a real number between **0** and **1** that defines the scale factor for the picture. The picture is scaled in such a way that

$$\frac{\text{width of picture}}{\text{width of help window}} = ratio$$

filename specifies the name of a file that contains a description of the picture. The file must be produced by the **xfig** version 3.1. The format of xfig files is given in appendix A. Here is the list of the fields ignored by the help system. Numbers refer to appropriate parts of the description.

2 **orientation**, **justification**, **units**, **coordinate_system**.

3 Colors with values greater than 7. Fill styles except -1 and 0.

3.1 Color pseudo-objects.

3.2 ARC: **sub_type**, **pen_style**, **style_val**, **cap_style**, **forward_arrow**, **backward_arrow**.

3.3 COMPOUND: all fields.

3.4 ELLIPSE: **sub_type**, **pen_style**, **style_val**, **direction**, **angle**, **start_x**, **start_y**, **end_x**, **end_y**.

3.5 POLYLINE: **sub_type**, **pen_style**, **style_val**, **join_style**, **cap_style**, **radius**, **forward_arrow**, **backward_arrow**.

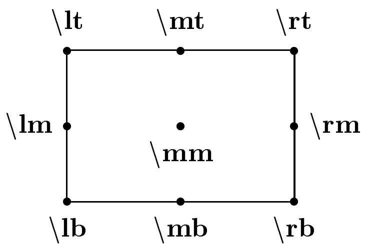
3.6 SPLINE: **pen_style**, **area_fill**, **style_val**, **cap_style**, **forward_arrow**, **backward_arrow**.

3.7 TEXT: **sub_type**, **pen_style**, **angle**, **font_flags** other than 1, **font** other than -1, 0, 1, 2, 3, 32.

The texts in TEXT objects may be preceded by one of the following commands:

\lt	**\mt**	**\rt**
\lm	**\mm**	**\rm**
\lb	**\mb**	**\rb**

These commands specify which point of the bounding rectangle should stay in its place during scaling. Graphic objects and texts are scaled by different factors because of 'discreteness' of available font sizes. The commands indicate points of bounding rectangles as shown on the diagram:



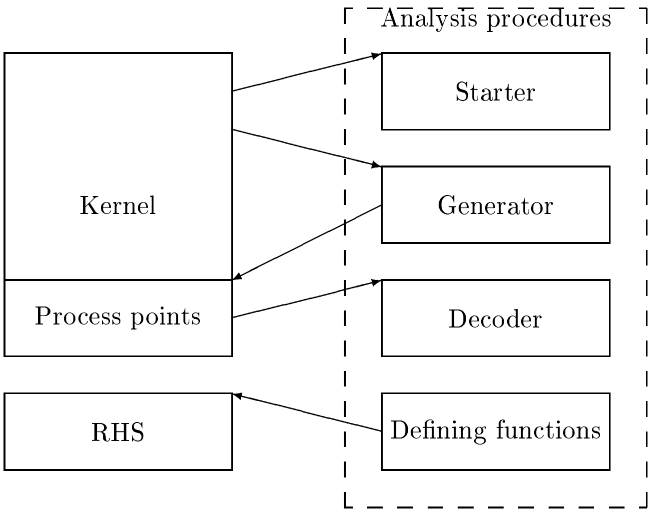
Chapter 7

Interface between kernel and analysis procedures

This chapter describes interface between the kernel and the analysis procedures. Interface between different components of analysis procedures is not fixed and left to the developer.

7.1 Calls diagram

The following diagram shows calls of analysis procedures made by the kernel. Is also shows calls to the functions that specify dynamical systems. Although these functions are not a part of the kernel and are not called by it they are compiled and loaded by the kernel. Components of analysis procedures that do not call the kernel and are not called by it are not shown.



7.2 Definition of entry points of analysis procedures

Two macros from the **common.h** file are used to define exported functions in dynamical objects (starters and generators in particular) and pointers to them.

The **Entry** macro is used to specify the type of a function with the attributes specific to particular operating system and/or compiler. The macro has one argument and is used at the place of the type of the function:

Entry(*type***)**

where *type* is the type of the function. It may be one of standard types (**void**, **int**, **double**, etc.) or one of defined in the Vibrant header files (**Int2**, **FloatHiPtr**, etc.). Do not add any class specifiers or OS-specific keywords like **extern** or **_export**, the macro already contains them as needed. An example of how the macro should be used:

Entry(double) add(double a, double b) ...

The **EntryPtr** macro is used to define pointers to exported functions. It has three arguments:

EntryPtr(*type*,*ptr*,(*parameters*)**)**

where *type* is the type of the function, *ptr* is the name of the pointer and *parameters* is comma separated list of parameters. For example, the following defines **ErrProcPtr** as a type of pointers to void functions with two arguments:

typedef EntryPtr(int,ErrProcPtr,(char *msg, int beep));

7.3 Calling conventions kernel-starter and kernel-generator

The types, macros and structures specifying the interface kernel-starters and kernel-generators are in the **stagen.h** file. Calling conventions for starters and generators are the same. Relevant types, macros and data structure from the **stagen.h** file are described below. Starters and generators should not access or modify other fields.

7.3.1 Types and macros

```
typedef EntryPtr(int,IntFuncEntry,());
```

The type of pointers to functions that compute the values of right-hand sides of dynamical systems. Therefore, all RHS should be of type **int** (3.2.7).

```
typedef EntryPtr(FloatHiPtr,DerFuncEntry,());
```

The type of pointers to functions that compute the values of partial derivatives of the functions specifying dynamical systems. The return value should be a pointer to the array containing the values of derivatives.

```
typedef EntryPtr(Int2,StagenPtr,(VoidPtr stagendata));
```

The type of pointers to starters and generators. The return values indicate success (0) or failure otherwise.

```
typedef EntryPtr(Int2,FuncParamPtr,());
```

The type of pointers to the following functions mentioned in curve type and class files:
defining functions (4.7),
test functions (4.8),
processing functions (4.9),
adapters (4.10),
functional parameters of starters (4.5),
functional parameters of generators (4.6),
standard linear algebra (3.2.5),
utility functions (3.2.6).

```
typedef EntryPtr(Int2,DecoderPtr,(VoidPtr point,
                                  FloatHiPtr PNTR indirect,
                                  Int2 oper));
```

The type of pointers to decoders (4.11). A point produced by a generator is pointed to by the **point** argument. Second argument, **indirect**, is a pointer to arrays where the decoder should put coordinates grouped in accordance with the definition of the visual classes of names (1.3.4). The **oper** argument is a number that specifies the action the decoder should take:

DECODER_INIT	initialize the decoder,
DECODER_TERM	terminate the decoder,
DECODER_DIM	return the number of mathematical points in ***point**,
$i \geq 0$	extract i-th mathematical point.

```
typedef EntryPtr(void,AuxFuncPtr,());
```

The type of pointer to auxiliary functions specifying the dynamical systems (e.g. boundary conditions).

```
typedef Uint2 Pointtype, PNTR PointtypePtr;
```

The type of points. Zero value means 'ordinary point'. The codes are listed in appropriate class files (3.2.1).

```
USERPOINT
```

The mask used by generators to mark points where one or more user functions vanishes (1.2.2).

```
UDF_IGNORE, UDF_DETECT, UDF_MONITOR, UDF_APPEND
```

The constants describing for generators how to process user functions (1.2.2).

```
typedef Int2 (PNTR CallbackPtr)(Pointtype type,
                                VoidPtr point,
```

```
                                        CharPtr msg);
```

The type of pointers to the function called by the generators to process produced points.

```
typedef Int2 (PNTR CallbackGetIndexPtr)(VoidPtr DataArea,
                                        Uint2 Offset);
```

The type of pointers to the function called by the starters and generators to obtain indices of fields in their data structures (Chapter 5). The indices are used for referencing the fields in update requests. The first argument must be the address of the data area of the starter of generator; it is passed as the value of the StaPtr or GenPtr field of the interface structure (7.3.2). **Offset** is the offset of the field from the beginning of the data area.

```
typedef void (PNTR CallbackUpdateIndexPtr)(Int2 Index);
```

The type of pointers to the function called by the starters and generators to update the values in their parameter windows (Chapter 5). The only argument is the index returned by the appropriate call to the function described above.

```
typedef int (PNTR CallbackUserFuncPtr)(VoidPtr vector,
                                       Int2 num,
                                       FloatHiPtr value);
```

The type of pointers to user functions. The functions are called by the generators to monitor their values, detect zeros, etc. (1.2.2). The first argument is a point computed by the generator. The second argument specifies particular function; it must be between **0** and the value of the **udFuncNum** field, inclusive.

```
typedef CharPtr (PNTR CallbackUserFuncNamePtr)(Int2 num);
```

The type of pointers to the function returning the names of user functions. Used by the generators to form appropriate messages. The argument specifies desired function.

```
RF_DO, RF_CLEAN, RF_CRASH
```

The flags indicating the reason of call of a starter of generator. Their meanings are as follows:

RF_DO	Normal call.
RF_CLEAN	(Starters only) Normal termination, free all allocated memory.
RF_CRASH	Crash, free all allocated memory.

```
#define ind2_(i,j)       ((i)+(((j)*((j)+1))>>1))
#define ind3_(i,j,k)     ((i)+(((j)*((j)+1))>>1)+ \
                          (k)*((k)+1)*((k)+2)/6)
#define ind4_(i,j,k,l)   (ind3_(i,j,k)+ \
                          (l)*((l)+1)*((l)+2)*((l)+3)/24)
#define ind5_(i,j,k,l,m) (ind4_(i,j,k,l)+(m)*((m)+1)*((m)+2)* \
```

```
((m)+3)*((m)+4)/120)
```

These macros map the ordinal numbers of variables with respect to which the partial derivatives of the orders **2**, **3**, **4**, and **5** are computed, respectively, to appropriate offsets in the matrices containing the values of the derivatives. The variables are numbered from zero and are in the order specified in the **&visiblenames** section of the specific class file (basic classes only, see 3.2.9). See the **_symdiff.c** file for an example of usage of these macros.

7.3.2 Data structure

When the kernel calls a starter or a generator it passes to them a pointer to the structure that contains pointers to all needed functions and data:

```
typedef struct _StagenData {
. . . . . . . . . . . . . . . . .
} StagenData, PNTR StagenDataPtr;
```

The relevant fields of the structure are described below. Starters and generators should not use or change other fields of the structure.

`VoidPtr StaPtr;`

The pointer to the data area of the starter formed in accordance with its parameter file (5.1).

`VoidPtr GenPtr;`

The pointer to the data area of the generator formed in accordance with its parameter file (5.1).

`IntFuncEntry Rhs;`

The pointer to the function computing the values of RHS. The calling conventions are described in appropriate class file (3.2.7).

`DerFuncEntry Der1, Der2, Der3, Der4, Der5;`

The pointers to functions that compute the values of derivatives of the orders 1–5 of the RHS. The calling conventions are described in appropriate class file (3.2.7).

`AuxFuncPtr PNTR AuxFunc;`

The pointer to the array of pointers to all the functions that specify the dynamical systems of the given class. The number of functions and calling conventions for them are given in the **&functions** section of the class file (3.2.7). The RHS and their derivatives may also be accessed through the pointer.

`CallbackPtr ProcessPoint;`

The pointer to the function called by the generators and processing functions to process produced points.

```
CallbackGetIndexPtr AskParIndex;
#define ParIndex(dataptr,field) AskParIndex(dataptr, \
        (CharPtr)&(dataptr->field)-(CharPtr)dataptr)
#define AllParIndices(dataptr) \
        AskParIndex(dataptr,(Uint2)0xFFFF)
```

The pointer to the function called by the starters and generators to obtain indices of fields in their data structures (Chapter 5). The indices are used for referencing the fields in update requests. The macro **ParIndex(***dataptr*,*field***)** is defined to ease calls of the function. The **AllParIndices(***dataptr***)** macro gets indices for all fields at one call.

```
CallbackUpdateIndexPtr UpdatePar;
```

The pointer to the function that updates specified field or all fields in starter's of generator's parameter window. The only argument must be obtained from a previous call to **AskParIndex**.

```
Int2Ptr DetectedTypes;
```

The pointer to the array of global (class-scope) codes of special points detectable on the curves of the given type. The array maps local codes used by generators and implicitly defined in the **&types** sections of the curve type files (4.2) to global codes defined in the class file (3.2.1): **DetectedTypes**[i] is the global code of the i-th class in the list (i=1,2,...). **DetectedTypes[0]** is always zero.

```
Int2Ptr PNTR ClassDim;
```

The pointer to the array of pointers to dimensions of the classes of visible names. The order of classes is the same as in the **&names** of the appropriate curve type file (4.3). **ClassDim[0]** is not used.

```
Uint2Ptr ClassReal;
```

The pointer to the array of global (class-scope) codes of the classes of visible names. The array maps local codes used by starters and implicitly defined in the **&names** sections of the curve type files (4.2) to global codes defined in the class file (3.2.9). **ClassReal[0]** is always zero.

```
Boolean Forward;
```

The direction indicator. It is **TRUE** if the user chose **Compute**|**Forward**. It is **FALSE** if the user chose **Compute**|**Backward**.

```
FuncParamPtr PNTR FuncParams[NKFP+2];
Int2 FuncParamsNum[NKFP+2];
```

The pointer to the array of pointers to functions passed to starters and generators. The **FuncParamsNum** gives the numbers of pointers. These two arrays are filled in based on descriptions from the curve type files. The following macros are defined to ease access to the functions. Comments refer to appropriate sections of class and curve type files.

```
#define NKFP         6
#define _DEF_INDEX   0
#define _TEST_INDEX  1
#define _STA_INDEX   2
#define _GEN_INDEX   3
#define _ADAPT_INDEX 4
#define _PROC_INDEX  5
#define _STDLINALG_INDEX NKFP
#define _UTILITIES_INDEX NKFP+1
/* &deffuncs - curve type file */
#define DefFunc      FuncParams[_DEF_INDEX]
#define DefFuncNum   FuncParamsNum[_DEF_INDEX]
/* &testfuncs - curve type file */
#define TestFunc     FuncParams[_TEST_INDEX]
#define TestFuncNum  FuncParamsNum[_TEST_INDEX]
/* &stafuncs - curve type file */
#define StaFunc      FuncParams[_STA_INDEX]
#define StaFuncNum   FuncParamsNum[_STA_INDEX]
/* &genfuncs - curve type file */
#define GenFunc      FuncParams[_GEN_INDEX]
#define GenFuncNum   FuncParamsNum[_GEN_INDEX]
/* &adapters - curve type file */
#define AdapterFunc  FuncParams[_ADAPT_INDEX]
#define AdapterNum   FuncParamsNum[_ADAPT_INDEX]
/* &procfuncs - curve type file */
#define ProcFunc     FuncParams[_PROC_INDEX]
#define ProcFuncNum  FuncParamsNum[_PROC_INDEX]
/* &stdlinalg - class file */
#define StdLinAlg    FuncParams[_STDLINALG_INDEX]
#define StdLinAlgNum FuncParamsNum[_STDLINALG_INDEX]
/* &utilities - class file */
#define Utilities    FuncParams[_UTILITIES_INDEX]
#define UtilitiesNum FuncParamsNum[_UTILITIES_INDEX]
```

`Int1Ptr TestSingMap;`

The pointer to a matrix described in the **&types** section of the curve type file (4.2). The matrix is used by generators to detect singularities based of the values of the test functions. The elements of the matrix may have the following values only: **0**, **1** or **-1**.

`FloatHiPtr BifDataOutPtr;`

The pointer to memory with data attached by processing functions to special points. The memory must be allocated and filled by processing functions before call to **ProcessPoint**. Its length and structure may vary from call to call. The kernel (**ProcessPoint**) frees the memory before return to processing functions.

The archive manager writes the data to the archive and associates them with the current special point. When this point is chosen as the initial point for another curve the kernel reads data associated with the point and assigns a pointer to it to the **BifDataInPtr** field (described below).

`size_t BifDataOutLen;`

The length of the data associated with the current special point. It is set by appropriate processing function. For ordinary points the generator must set this field to zero. The **ProcessPoint** function clears the field before return.

`FloatHiPtr BifDataInPtr;`

The pointer to data associated with the currently chosen special point. The kernel allocates and fills the memory. It must be freed by the starter.

`size_t BifDataInLen;`

The length of the data associated with the currently chosen special point. It is set be the kernel before call of the starter.

`Boolean BifDataInOk;`

When this field contains **TRUE** the **BifDataInPtr** and **BifDataInOk** fields contain valid values. Otherwise their values should not be used.

`FloatHiPtr ContDataGenPtr;`

The pointer to memory allocated and filled by the generator before it returns. The data are used by the generator when it extends the curve. The archive manager writes the data to the archive and frees the memory. When the user chooses the curve the data are read and a pointer to the memory is assigned to this field allowing the generator to use the data.

`size_t ContDataGenLen;`

The length of the data pointed by the **ContDataGenPtr** field. Should be set by the generator before return. Set by the kernel when a curve is chosen either implicitly or explicitly.

`FloatHiPtr ContDataStaPtr;`

Similar to the **ContDataGenPtr** field but used by the starter. Usually the starter allocates and fills the memory when it is called for normal termination (the field **Reason** contains **RF_CLEAN**).

`size_t ContDataStaLen;`

Specifies the length of data pointed by the **ContDataStaPtr** field.

`Int1 SuspendIsPending;`

Used internally by the kernel during computations of the curves. It may have following values:

0 Normal mode, compute next point.
1 Suspend computation.
2 Abort computation.

`Int2 udFuncNum;`

The number of user functions. Disabled functions are not counted.

`CallbackUserFuncPtr udFunc;`

The pointer to the array of pointers to user functions. The number of pointers is given by the **udFuncNum** field.

`CallbackUserFuncNamePtr udFuncName;`

The pointer to the function that returns the names of the user functions. Used by the generators to construct messages.

`Int2 Reason;`

Indicates the reason of the call:

RF_DO indicates normal call. Starters and generators are called with this value after the **Compute** command.

RF_CLEAN is used for starters only. It means that the computation of the current curve has been successful and requires the starter to free all memory allocated in response to the previous **RF_DO** call.

RF_CRASH informs the starter and the generator that the computation of the current curve has been aborted due to an error and requires to free all memory allocated in response to the previous **RF_DO** call.

`struct _StagenData PNTR Slave;`

The pointer to the data structure build for the slave starter and generator. This pointer is nonzero only if the **&slave** section is present in the curve type file (4.12).

`StagenPtr pStarter;`

The pointer to the slave starter. This field contains NULL for primary starter.

`StagenPtr pGenerator;`

The pointer to the slave generator. This field contains NULL for primary generator.

`DecoderPtr pDecoder;`

The pointer to the slave decoder. This field contains NULL for primary decoder.

`LibHandle PNTR FuncParamsHandle[NKFP+2];`

Used internally by the kernel.

`FloatHiPtr PNTR IndirectValues;`
Used internally by the kernel.

`VoidPtr StaParDesc;`
Used internally by the kernel.

`VoidPtr GenParDesc;`
Used internally by the kernel.

7.3.3 The argument

Each starter and generator has one argument of the type **StagenDataPtr** described in 7.3.2.

When the kernel calls a starter and a generator in a normal way (i.e. for computing curves) the pointer passed is nonzero and the **Reason** field contains **RF_DO**. Starters and generators should copy the pointer to its own static variable with the file scope. It is used by various components of analysis procedures during computations of curves.

After the computation has been completed the kernel calls the starter and the generator again to let them to free memory. These calls are made with the NULL pointer as the parameter. The **Reason** field describes the outcome of computation: **RF_CLEAN** means normal termination and **RF_CRASH** means that the computation was aborted due to an error (exception while computing RHS, etc.). For normal termination the kernel assumes that the generator has already freed all the memory and does not call it.

An important consequence of this discipline is that the starters and the generators must keep references to all dynamically allocated objects in variables with the file scope. This would allow one to avoid dangling references when computation is aborted due to errors.

7.4 Calling conventions generator-kernel

Starters and generators use the **ProcessPoint** function to pass computed points to the kernel. They call the function via the pointer from the **StagenData** structure passed to them (7.3.2).

Starters should call the function only to let the kernel know the last point of a curve. When the starter is called with the **NULL** pointer and the **Reason** is equal to **RF_CLEAN** it must call the function with the last computed point and a parameter. This allows the archive manager to store all coordinates of the last point into the archive which in turn makes is possible to extend the curve later on. The type of the last point must be that of the computed curves. When the generator calls the function is cannot know whether it is the last point of a curve or not: at the pause after processing the point the user may choose to continue or to abort. That is why the default processor should keep a copy of the last computed point.

The **ProcessPoint** function should be called as follows (the **genData** is a copy of the pointer passed to the generator by the kernel):

```
genData->ProcessPoint(type,pt,msg);
```

where **type** specifies the type of the point and has one of the following values (see 4.2)

0	for ordinary points,
i	$1 \leq i \leq s$ for singular points,
$s+1$	for the last point.

pt is a pointer to the computed point. The kernel will call the Decoder (7.5) to extract values from it. **msg** is a string with a message the interface manager displays in the main window. It may contain a text describing special points. For ordinary points this parameter must be **NULL**.

7.5 Calling conventions kernel-decoder

A decoder is called by the kernel for each point got from a generator. Remind that the generator passes points it produces to the kernel via the **ProcessPoint** function pointer to which is in the **StagenData** data stricture described in 7.3.2.

Each decoder has three argument and should be defined as follows:

```
Entry(Int2) ep_Decoder(VoidPtr vpoint,
                        FloatHiPtr PNTR indirect,
                        Int2 oper) {
. . . . . . . . . . . . . . . . . . . . . . . . .
}
```

The first argument, **vpoint** is a pointer to the point computed by the generator. The **indirect** is a pointer to an array of pointers. For each class of visible names from the **&names** section of the curve type file the decoder must assign to appropriate element of the **indirect** array a pointer to values from this class extracted from the point **vpoint**. The third argument specifies requested action and may have one of the following values: **DECODER_INIT**, **DECODER_TERM**, **DECODER_DIM**, or an nonnegative integer. Their meaning was described in 7.3.1.

As an example let us consider the decoder that maps equilibrium points (the file **__ep.c**. Classes of visible names are represented by the following constants. Note that the values reflect the order in which classes are listed in the **__ep.con** file.

```
#define CL_TIME    1
#define CL_PHASE   2
#define CL_PAR     3
```

```
#define CL_EIGV    4
#define CL_TEST    5
#define CL_UTEST   6
#define CL_STDFUN  7
```

There are three starters for equilibrium curves. They differ from each other in the type of initial points. Namely, equilibrium curves may be computed from ordinary points (it is assumed, that the used gives some 'good' approximation), equilibrium points and branching points. Initial points of the last two types may be taken on previously computed curves. Each starter allocates several global vectors. We will consider only two of them, **X** and **P** used for values of the phase variables and parameters, respectively. Here is a sketch of how starters and the decoder work with the vectors.

```
/* global copy of Starter's parameter passed by content */
Local(StagenDataPtr) stagenData=NULL;
Local(FloatHiPtr) X=NULL;
Local(FloatHiPtr) P=NULL;
. . . . . . . . . . . .
Entry(Int2) ep_Starter(StagenDataPtr stagenptr) {
  stagenData=stagenptr;
  if (stagenptr) {  /* Normal computation, allocate vectors */
    /* Use standard linear algebra and utilities */
    staFunc=(StdLinAlgDataPtr)stagenptr->StdLinAlg;
    staUtil=(UtilitiesDataPtr)stagenptr->Utilities;
    /* Get dimensions of Phase and Par spaces */
    npar=*stagenptr->ClassDim[CL_PAR];
    nphase=*stagenptr->ClassDim[CL_PHASE];
    /* Get the number of active parameter */
    if (staUtil->CheckParameters(staData->Active,npar,
                             &active,1)) {
      active=MemFree(active);
      return 1; /* there must be one active parameter */
    }
    P=MemNew(sizeof(FloatHi)*npar);
    X=MemNew(sizeof(FloatHi)*nphase);
    . . . . . . . . . . . . . . . . .
  } else {          /* Termination, free memory */
    X=MemFree(X);
    P=MemFree(P);
  }
}
. . . . . . . . . . . .
Local(void) CopyToX(Vector vp) {
  MemCopy(X,vp,sizeof(FloatHi)*nphase);
}
```

```
  Local(void) CopyToP(Vector vp) {
    P[active]=vp[nphase];
 }
 . . . . . . . . . . . . .
 Entry(Int2) ep_Decoder(VoidPtr vpoint,
                        FloatHiPtr PNTR indirect,
Int2 oper) {
   switch (oper) {
     case DECODER_INIT:
     case DECODER_TERM:
       break;
     case DECODER_DIM:
       return 1; /* always one mathematical point */
     default:    /* extract M-point */
       indirect[CL_PHASE]=X;
       CopyToX(vpoint);
       indirect[CL_PAR]=P;
       CopyToP(vpoint);
       . . . . . . . . . .
 }
 return 0;
}
```

Note that points produced by the generator have **nphase+1** values, **nphase** phase components and one for the active parameter.

7.6 Calling conventions of functions specifying dynamical systems

Calls to the functions specifying dynamical systems are described in the **&functions** section of the corresponding class file (3.2.7). They should be called accordingly. The kernel itself does not put any constraints on the calling conventions.

RHS and their derivatives are available via the following fields of the **StagenData** structure described in 7.3.2: **Rhs**, **Der1**, **Der2**, **Der3**, **Der4**, and **Der5**. The **AuxFunc** field provides access to all functions specifying the dynamical systems. The number and order of pointers are the same as in the **&functions** section.

Before calling the derivatives starters and generators should first check corresponding pointers. If a pointer is nonzero then there is a function that symbolically computes the values. Otherwise, the numerical approximation must be made. This choice for the derivatives is made by the user in the System specification dialog box.

Chapter 8

Adding new analysis procedures

CONTENT allows the developer to add new analysis procedures. There may be two different situations when the developer may want to "extend" CONTENT :

- support new curve type for already supported class of dynamical systems.
- add support of new class of dynamical systems.

In both situations appropriate analysis procedures must be written and some of configuration and data files created and modified. Note that this does not require any modification, compilation or linkage of the kernel and existing analysis procedures.

8.1 Supporting of new curve types

Supporting of new curve type requires the following steps to be taken.

1. Develop and implement analysis procedures in the **C** programming language (described in 8.3 and Chapter 7).
2. Create the parameter file for the starter (Chapter 5). Create the parameter file for the generator if you plan to develop new generator. However, you may want to use one of the generators supplied with CONTENT (the iterator, integrators, or the continuer).
3. Create appropriate curve type file that specifies data needed to access analysis procedures (Chapter 4).
4. Modify appropriate class file (3.2). Namely, lines should be added (or modified) to the following sections of the file:

&graph - descriptions of new type of points and types of curves that may be computed from points of this type (3.2.1).

&compclass - the class of the generator that computes the curves of the new type (3.2.2).

&menu - items for the **Type|Initial point** and **Type|Curve** sub menus (3.2.3).

&deftypes - default curve type set for initial points of the new type (3.2.4).

&derivatives - specifies the maximum order of derivatives needed for computations of curves; increase it if necessary (3.2.8).

&visiblenames - description of a class of visible names for the test functions used to detect singularities on the curves of new type (3.2.9).

&help - specify the name of the help file that contains topics related to new type of curves (3.2.10, Chapter 6).

5. Prepare source help file with topics describing new curve type and process it by the help compiler (Chapter 6).

It is recommended to use supplied configuration and data files as well as sources of analysis procedures as templates and examples.

8.2 Supporting of new classes of dynamical systems

Supporting of new classes of dynamical systems requires the following steps to be taken.

1. Modify the main class files adding the description of new class to it (3.1).
2. Prepare source help file with topics that give mathematical description of the dynamical systems belonging to the class as well as all aspects of their specifications and process it by the help compiler (Chapter 6).
3. Create appropriate class file (3.2).
4. For each type of curves you plan to support for new class follow the steps listed in 8.1.

8.3 Development of analysis procedures

For each curve type the following analysis procedures should be implemented:

- Starter (one starter for each possible type of initial points),

- Defining functions,
- Test functions,
- Processing functions,
- Adapter,
- Decoder,
- Generator (unless one of the supplied generators can be used).

Note again that each possible combination of types of initial points and curves require implementation of specific starter. For example (see the **ode_g.con** file), equilibria curves may be computed from points of these types: **O**, **EP**, **BP**, **LP**, and **H**. Therefore, five starters are needed for the **EP** curve type; they compute initial points of equilibria curves depending on the type of initial point chosen by the user.

8.3.1 General recommendations

All analysis procedures listed in 8.3 except generators usually share some data. For example, results of calculation of defining functions may be used by test functions. Or the adapter may change vectors used by the defining functions. That is why it is recommended to put all these functions in one source file.

Execution of analysis procedures may be interrupted due to exceptions at any moment (examples are division by zero, square root of negative numbers, invalid access to memory). This is especially true for the defining functions which call RHS. In a case of any such event CONTENT terminates the computation of the current curve and informs the user. The problem now is: how to free all the memory allocated by the starter and the generator. The solution is to keep addresses of all dynamically allocated pieces of memory in static pointers with the file scope (i.e. defined outside of functions). The kernel calls the starter and the generator with a special indicator of an error (the **Reason** field has one of the values **RF_CLEAN** or **RF_CRASH**, see 7.3.2). Keeping the references in non-local pointers allows them to free allocated memory. Note that memory can be allocated by one of the memory allocation routine or by functions from linear algebra packages (8.3.2).

It is recommended to give unique names to analysis procedures. This can be done by prefixing or suffixing names by labels of curve types (1.3.3). For example, starters that compute initial points for the equilibria curves have names **o_ep_Starter**, **ep_Starter**, and **bp_ep_Starter** for initial points of types **O**, **EP**, and **BP**, respectively.

8.3.2 Access to functions needed by analysis procedures

Analysis procedures need access to many general-purpose functions that are provided by CONTENT. Starters and generators should always use these functions instead of the "standard" ones to provide portability. These functions may be classified as follows:

Core functions Provide memory allocation, string manipulation, file input-output. Header file is **corefunc.h**.

Utility functions Perform various checks needed by starters. Header file is **_util.h**.

Linear algebra functions Provide class specific vector and matrix allocation and manipulation. Header files are **_linalg.h** (standard package, see the **&stdalg** section in class files), **_rdalg** (used for PDEs), **_bvpalg** (used for limit cycles of ODEs).

Functions specifying dynamical system These functions calculate the values of RHS, boundary conditions, etc.

The way of calling a function depends on the category the function belongs to.

Core functions should be called by their names. The header file, **corefunc.h**, provides all necessary definitions. In addition it declares the pointer, **coreFuncPtr**, to the table with pointers to all functions and appropriate **#define** directives. Core functions are linked with the kernel; it provides the pointer to the table. See the header file for details.

Many core functions are from the **Vibrant**[1] library on which CONTENT is based. They provide portability across different hardware and software platforms.

The following core functions may be called by starters and generators.

```
VoidPtr MemNew(size_t size);
```

Allocates *size* bytes of memory and initializes them to zeros.

```
VoidPtr MemGet(size_t size, unsigned int flags);
```

Allocates *size* bytes of memory. Initializes them to zeros if *flags* is **MGET_CLEAR**. Another legal value of *flags* is zero in which case memory is not filled.

```
VoidPtr MemMore(VoidPtr ptr, size_t size);
```

Increases or decreases the size of a previously allocated block of memory. *ptr* must be a value obtained from a previous call to one of allocation routines; it should not by **NULL**. Additional bytes are not initialized.

```
VoidPtr MemExtend(VoidPtr ptr, size_t size,
                                    size_t oldsize);
```

[1]**Vibrant** was developed by J. Kans and his colleagues at National Center for Biotechnology Information, National Library of Medicine, National Institute of Health, USA. It is available via an anonymous ftp (**ncbi.nlm.nih.gov**, **130.14.20.1**). Cd to **toolbox**, cd to **ncbi_tools**, set **bin** mode and get appropriate file.

Increases or decreases the size of a previously allocated block of memory. *ptr* must be a value obtained from a previous call to one of allocation routines; it should not by **NULL**. Additional bytes are initialized to zeros.

```
VoidPtr MemFree(VoidPtr ptr);
```

Frees a previously allocated block of memory. Does nothing if *ptr* is **NULL**. Always returns **NULL**. For safety, always assign the result to the pointer: *ptr*=**MemFree(***ptr***)**

```
VoidPtr MemFill(VoidPtr ptr, int value, size_t bytes);
```

Fills a range of bytes with a specified value.

```
void * MemSet(void *ptr, int value, size_t bytes);
```

The same as **MemFill**

```
void * MemCopy(void *dst, const void *src, size_t bytes);
```

Copies a range of bytes. Should not be used for overlapping regions.

```
void * MemMove(void *dst, const void *src, size_t bytes);
```

Copies a range of bytes. Works for overlapped regions.

```
size_t StrLen(const char *str);
size_t StringLen(const char *str);
int StrCmp(const char *s1, const char *s2);
int StrICmp(const char FAR *a, const char FAR *b);
CharPtr StrChr(const char *s, int c);
CharPtr StringChr(const char FAR *str, int chr);
CharPtr StrStr(const char *s1, const char *s2);
CharPtr StringStr(const char *s1, const char *s2);
CharPtr StrCpy(char *s1, const char *s2);
CharPtr StrNCpy(char *s1, const char *s2, size_t n);
CharPtr StringNCpy(char *s1, const char *s2, size_t n);
CharPtr StrPBrk(const char *s1, const char *s2);
CharPtr StrCat(char *s1, const char *s2);
CharPtr StringCat(char *s1, const char *s2);
CharPtr StringTok(char FAR *str1, const char FAR *str2);
```

These functions are similar to their ANSI C namesakes. Functions that begin with 'String' accept **NULL** string pointers (in such cases they do nothing but return).

```
CharPtr StringSave(const char FAR *from);
```

Allocates memory and copies a string given by *from* to it. Returns the pointer to allocated memory. Memory should be freed by **MemFree** later.

```
void StrTrim(CharPtr source);
```

Removes leading and trailing white spaces.

```
CharPtr StrSubstitute(CharPtr source, CharPtr what,
                      CharPtr with, Boolean wholeid,
                      Int2Ptr num);
```

Replaces all occurrences of the string pointed to by *what* by another string pointed to by *with*. If *wholeid* is **TRUE** replacement takes place for "maximal" identifiers only (i.e. when an occurrence of ***what** does not has alphanumeric characters on the left and on the right). If *num* is nonzero it is assigned the number of replacement made by the function. This function may allocate memory for the target string via **MemNew**. In such a case it uses **MemFree** to free the source string.

```
CharPtr StrInsert(CharPtr source, Int2 position,
                  CharPtr what);
```

Inserts a string pointed to by *what* into a string pointed to by *source* at the given position. Always allocates memory for the result string and frees the source string.

```
ByteStorePtr BSNew(Int4 len);
ByteStorePtr BSFree(ByteStorePtr bsp);
Int4 BSWrite(ByteStorePtr bsp, VoidPtr ptr, Int4 len);
Int2 BSPutByte(ByteStorePtr bsp, Int2 value);
Int4 BSRead(ByteStorePtr bsp, VoidPtr ptr, Int4 len);
Int2 BSSeek(ByteStorePtr bsp, Int4 offset, Int2 origin);
Int4 BSTell(ByteStorePtr bsp);
Int4 BSDelete(ByteStorePtr bsp, Int4 len);
VoidPtr BSMerge(ByteStorePtr ssp, VoidPtr dest);
```

These **Vibrant** functions provide manipulations of so called ByteStores. A ByteStore is a linear sequence of bytes of memory that can be dynamically (re)allocated, read, written, positioned, etc. in the similar way to file input-output functions. See **Vibrant** documentation for more details.

```
MsgAnswer Message(Int2 keysev, const CharPtr fmt,...);
```

Displays a message in a dialog box. *keyserv* specifies the buttons in the dialog box. It may be **MSG_OK** (one **Ok** button), **MSG_OKC** (**Ok** and **Cancel** buttons) or **MSG_YN** (**Yes** and **No** buttons. *fmt* and other arguments are used to form the message in the same way as the **printf** function does. The return value may be **ANS_OK**, **ANS_CANCEL**, **ANS_YES**, or **ANS_NO** depending on the button the user presses.

```
void Beep(void);
```

Issues a short beep.

```
void DayTimeStr(CharPtr buf, Boolean date,
                Boolean time);
```

Fills in the *buf* with current data (if *date* is **TRUE** and/or time (if *time* is **TRUE**. The *buf* should be at least 24 bytes long.

```
FILE *fopen(const char *filename, const char *mode);
int fclose(FILE *stream);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *s, const char *format, ...);
int printf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
int fseek(FILE *stream, long offset, int whence);
void rewind(FILE *stream);
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
int fputs(const char *s, FILE *stream);
long ftell(FILE *stream);
CharPtr fgets)(char *s, int n, FILE *stream);
int fgetc(FILE *stream);
```

Standard file input-output functions.

```
const Char * TSPrintf(const Char * fmt, ...);
const Char * TSPrintfArgs(const Char * fmt, va_list args);
```

These **Vibrant** functions are similar to **sprintf** and **vsprintf** with the exception that the functions allocate and return buffers large enough to hold formatted strings. Next call to the functions overwrites the buffer. You should never free this memory.

```
void qsort(void *base, size_t nmemb, size_t size,
           int (_USERENTRY *compar)(const void *, const void *));
void qsort(void *base, size_t nmemb, size_t size,
           EntryPtr(int,compar,(const void *, const void *)));
```

Standard **qsort** function. The first version should be used in Windows. The second one is for Unix. Use the macros **_WIN** and **_UNIX** defined in the **common.h** file to implement OS-dependent pieces of code:

```
#if _WIN
  /* Windows */
  . . . . . . .
#endif
#if _UNIX
  /* Unix */
  . . . . . . .
#endif
```

```
double atof(const char *s);
```

Standard **atof** function.

```
CharPtr GetParamString(CharPtr section, CharPtr key);
void SetParam(CharPtr section, CharPtr key, CharPtr value);
```

These functions provide access to the settings file (2.1). *section* and *key* specify section and key, respectively. *value* gives new value for the *key*.

To use utility functions include appropriate header, **_util.h**. It declares all utility functions and the structure, **UtilitiesData**, which contains pointers to the functions. The kernel fills in the structure and passes the pointer to it via the **Utilities** field (7.3.2). There is a sketch of how one can call utility functions.

```
Local(UtilitiesDataPtr) util=NULL;
. . . . . . . . . . . . . . . . .
Entry(Int2) Starter(StagenDataPtr stagen) {
  util=(UtilitiesDataPtr))stagen->Utilities;
  /* now call a utility function */
  util->ErrMessage("Error");
  . . . . . . . . . . . . . . . . . . . . . .
}
```

There are following utility functions.

```
CheckParameters(CharPtr Active, Int2 ParNum,
                Int2Ptr Indices, Int2 ActiveNum);
```

This function checks that the number of active parameters in the starter's parameters window is equal to the given number, *ActiveNum*. The subscripts of active parameters will be written to the *Indices* array that must have at least *ActiveNum* elements. *ParNum* specifies the total number of parameters of dynamical system. *Active* is a vector indicating the activity state. This should usually be taken from the starter's data area (5.2.3). All starters for bifurcation curves have appropriate fields called **Active**. See for example the **__ep.h** file.

```
void ErrMessage(CharPtr msg);
```

Displays a dialog box with a message specified by *msg*. Returns after the user presses the **Ok** button in the dialog box.

```
void ActivateTestFuns(StagenDataPtr stagen, BoolPtr asg,
                      BoolPtr atf);
```

This function is used by the continuer to find out which test functions should be monitored based on the set of singularities chosen by the user (the user highlights the singularities he/she wants to be detected). *stagen* should be the value passed by the kernel to the continuer. *asg* is a vector of boolean values indicating chosen singularities. It is the value of the **ActiveSing** field from continuer's data area; it is filled by starters. *atf* is a boolean vector allocated by the continuer. The function assigns **TRUE** to components that correspond to test functions whose values should be monitored.

```
void IndProcFunc(StagenDataPtr stagen, BoolPtr asg,
                 Int1Ptr itf, BoolPtr cs);
```

This function is used by the continuer to detect singularities based on the values of the test functions. *stagen* should be the value passed by the kernel to the continuer. *asg* is a vector of boolean values indicating chosen singularities. It is the value of the **ActiveSing** field from continuer's data area; it is filled by starters. *itf* is a vector indicating the values of test functions according to the following table:

0	zero
1	nonzero
-1	irrelevant

Note that this function uses data from the **&types** section of the curve type file (4.2).

Access to functions from linear algebra packages is provided via appropriate pointers passed by the kernel: **StaFunc**, **GenFunc** and **StdLinAlg**. Calling conventions of functionally equivalent functions are kept the same. This allows for generators to use different linear algebra packages hiding the differences in the curve type file. Generators usually include the **_linalg.h** header file. Starters should include appropriate header file as they usually exploit functions specific for the package. The header files define the structure of pointers to all functions from packages. Here is an example of access to functions:

```
#include "_linalg.h"   /* defines StdLinAlgDataPtr */
Local(StdLinAlgDataPtr) staFunc=NULL;
Local(Vector) x0=NULL; /* Vector is defined in _linalg.h */
Local(Int2) npar;
. . . . . . . . . . . . . .
Entry(Int2) Starter(StagenDataPtr stagenptr) {
  staFunc=(StdLinAlgDataPtr)stagenptr->StdLinAlg;
  npar=*stagenptr->ClassDim[CL_PAR]);
  staFunc->CreateVector(npar);
  . . . . . . . . . . . . .
}
```

The calling conventions of the functions specifying dynamical systems depend on specific class the systems belong to. The generators should not call these functions directly. Instead they should call defining and test functions. This provides independence of generators from specific types of computed curves. Other analysis procedures should call the functions as described in 7.3.2 (the pointers **Rhs**, **Der1...Der5** and **AuxFunc**) and 3.2.7.

8.3.3 Technical issues

Many aspects of designing of analysis procedures are given in previous chapters. Here are some additional issues that should be taken into account by the developer. Note that there are many more issues left beyond this document: namely those that are related to internal structure and/or behavior of the analysis procedures.

1. If the parameter, *stagenptr*, passed by the kernel to the starters or the generators is **NULL** then the starter or the generator has to free all memory allocated during previous call and return. To distinguish between normal and abnormal termination the value of the **Reason** field may be tested (7.3.2).

2. To provide 'extending' of computed curves starters and generators may associate some data with them. This has to be done before return (for generators) or in response to the **RF_CLEAN** request (for starters). The length of data should be assigned to the **ContDataStaLen** and **ContDataGenLen** fields, respectively. Allocate memory via calls to **MemNew** and assign the pointers the the **ContDataStaPtr** and **ContDataGenPtr** fields. There are no restrictions on the length or contents of data.

3. The starter should distinguish whether the new curve has to be computed or the current one 'extended':

   ```
   if (stagenptr->ContDataGenLen) {
     /* Extend the current curve */
     /*
        stagenptr->ContDataStaPtr points to
        data associated with the curve,
        stagenptr->ContDataStaLen is the length of data
     */
     . . . . . . . . . . . . . . . .
     stagenptr->ContDataStaPtr=
                MemFree(stagenptr->ContDataStaPtr);
     stagenptr->ContDataStaLen=0;
   } else {
     /* Compute new current curve */
     . . . . . . . . . . . . . . . .
   }
   ```

   Note that data associated with the curve must be freed and the length set to zero before return from the starter and generator.

4. The starter should allocate and fill in some vectors for the generator. The number and semantics of vectors depend on the generator used to compute curves of the given type. For example, the continuer needs two vectors: a vector with the coordinates of the initial point and a tangent vector at the point:

   ```
   Local(Vector) x0=NULL,v0=NULL;
   ```

```
Entry(Int2) Starter(StagenDataPtr stagenptr) {
  if (stagenptr) { /* Normal call, proceed */
    if (stagenptr->ContDataGenLen) {
      . . . . . . . . . . . . . .
    } else { /* New curve */
      x0=staFunc->CreateVector(nphase);
      v0=staFunc->CreateVector(nphase);
      MemCopy(x0,staData->X,sizeof(double)*nphase);
      . . . . . . . . . . . . . .
      ((ContDataPtr)stagenptr->GenPtr)->X0=x0;
      ((ContDataPtr)stagenptr->GenPtr)->V0=v0;
    }
  } else { /* Termination */
    x0=staFunc->DestroyVector(x0);
    v0=staFunc->DestroyVector(v0);
  }
  return 0;
}
```

5. Zero return value from the starter is interpreted by the kernel as "no errors". In such a case the generator is called after the starter. In a case of an error (for example, iterations does not converge) the starter and the generator should free all allocated memory and return nonzero value. Note once more, for nonzero return values the kernel does not call the starter and the generator with **NULL** parameter and the **Reason** field set to **RF_CLEAN**.

Appendix A

Description of the xfig format

This appendix describes the format of the files created by xfig. The files are used to produce pictures (6.13). The following description is the contents of the **Doc/FORMAT3.1** file found in the xfig documentation. Note that the help systems supports only limited number of features described here.

```
/*
 * FIG : Facility for Interactive Generation of figures
 * Copyright (c) 1985 by Supoj Sutanthavibul
 * Parts Copyright (c) 1994 by Brian V. Smith
 * Parts Copyright (c) 1991 by Paul King
 *
 * The X Consortium, and any party obtaining a copy of these files from
 * the X Consortium, directly or indirectly, is granted, free of charge, a
 * full and unrestricted irrevocable, world-wide, paid up, royalty-free,
 * nonexclusive right and license to deal in this software and
 * documentation files (the "Software"), including without limitation the
 * rights to use, copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit persons who receive
 * copies from any such party to do so, with the only requirement being
 * that this copyright notice remain intact.  This license includes without
 * limitation a license to do the foregoing actions under any patents of
 * the party supplying this software to the X Consortium.
 */

The only difference from version 3.0 to version 3.1 is that the position
of the "magnet" has been shifted by 14 Fig units.
In the 2.1 and older versions of xfig the grid was in multiples of 5 Fig
units, but they were on intervals 4, 9, 14, 19, etc.
When version 3.0 was created, coordinates were simply multiplied by the
ratio of the resolutions (1200/80 = 15) so values like 4 became 60 instead
of 74 ((4+1)*15 - 1).

This means that figures converted from 2.1 and older files are offset by
14 Fig units but new objects entered with version 3.0 are correct.
```

```
In version 3.1 the magnet grid is at intervals 0, 75, 150, etc instead of
-1, 74, 149, etc.
Figures from 2.1 and older are correctly converted now and a warning is popped
up when you read in a version 3.0 file that says you may have to offset the
figure when you load it, using the x and y offsets in the file panel.

-------------------------------------------------------------------------------
Description of the Fig Format Follows
-------------------------------------------------------------------------------

(1) The very first line is a comment line containing the name and version:
#FIG 3.1

    The character # at the first column of a line indicates that the line
    is a comment line which will be ignored.

(2) The first non-comment line consists of two numbers and two strings:

int fig_resolution (Fig units/inch)
string orientation ("Landscape" or "Portrait")
string justification ("Center" or "Flush Left")
string units ("Metric" or "Inches")
int coordinate_system (1: origin is the lower left corner (NOT USED)
 2: upper left)

    Fig_resolution is the resolution of the figure in the file.
    Xfig will always write the file with a resolution of 1200ppi so it
    will scale the figure upon reading it in if its resolution is different
    from 1200ppi.  Pixels are assumed to be square.

    Xfig will read the orientation string and change the canvas to match
    either the Landscape or Portrait mode of the figure file.

    The units specification is self-explanatory.

    The coordinate_system variable is ignored - the origin is ALWAYS the
    upper-left corner.

    ** Coordinates are given in "fig_resolution" units.
    ** Line thicknesses are given in 1/80 of an inch ("display units").  The
       minimum line thickness is 0 (no line is drawn) and the maximum is 500.
    ** dash-lengths/dot-gaps are given in 1/80 of an inch.


(3) The rest of the file contains various objects.  An object can be one
    of six classes (or types).

0) Color pseudo-object.
1) Arc.
2) Ellipse which is a generalization of circle.
3) Polyline which includes polygon and box.
4) Spline which includes closed/open control/interpolated spline.
5) Text.
6) Compound object which is composed of one or more objects.

    In the following elaboration on object formats, every value of Fig
    output are separated by blank characters or new line ('\n').  The
```

```
    value of the unused parameters will be -1.

    Some fields are described as "enumeration type" or "bit vector"; the
    values which these fields can take are defined in the header file object.h.
    The pen_style field is unused.
    These values may be defined in some future version of Fig.

    The two color fields (pen and fill; pen only, for texts) are
    defined as follows:

    -1 = Default
     0 = Black
     1 = Blue
     2 = Green
     3 = Cyan
     4 = Red
     5 = Magenta
     6 = Yellow
     7 = White
  8-11 = four shades of blue (dark to lighter)
 12-14 = three shades of green (dark to lighter)
 15-17 = three shades of cyan (dark to lighter)
 18-20 = three shades of red (dark to lighter)
 21-23 = three shades of magenta (dark to lighter)
 24-26 = three shades of brown (dark to lighter)
 27-30 = four shades of pink (dark to lighter)
    31 = Gold

 values from 32 to 543 (512 total) are user colors and
 are defined in color pseudo-objects (type 0)

    For WHITE color, the area fill field is defined as follows:

-1 = not filled
 0 = black
...  values from 1 to 19 are shades of grey, from darker to lighter
20 = white
21-40 not used
41-56 see patterns for colors, below

    For BLACK or DEFAULT color, the area fill field is defined as follows:

-1 = not filled
 0 = white
...  values from 1 to 19 are shades of grey, from lighter to darker
20 = black
21-40 not used
41-56 see patterns for colors, below

    For all other colors, the area fill field is defined as follows:

-1 = not filled
 0 = black
...  values from 1 to 19 are "shades" of the color, from darker to lighter.
A shade is defined as the color mixed with black
20 = full saturation of the color
...  values from 21 to 39 are "tints" of the color from the color to white.
```

```
A tint is defined as the color mixed with white
40 = white
41 = 30 degree left diagonal pattern
42 = 30 degree right diagonal pattern
43 = 30 degree crosshatch
44 = 45 degree left diagonal pattern
45 = 45 degree right diagonal pattern
46 = 45 degree crosshatch
47 = bricks
48 = circles
49 = horizontal lines
50 = vertical lines
51 = crosshatch
52 = fish scales
53 = small fish scales
54 = octagons
55 = horizontal "tire treads"
56 = vertical "tire treads"

    The depth field is defined as follows:

 0 ... 999 where larger value means object is deeper than (under)
   objects with smaller depth

    The line_style field is defined as follows:

-1 = Default
 0 = Solid
 1 = Dashed
 2 = Dotted

    The style_val field is defined as the length, in 1/80 inches, of the on/off
    dashes for dashed lines, and the distance between the dots, in 1/80 inches,
    for dotted lines.

    The join_style field is defined FOR LINES only as follows:

 0 = Miter (the default in xfig 2.1 and earlier)
 1 = Bevel
 2 = Round

    The cap_style field is defined FOR LINES, OPEN SPLINES and ARCS only as
    follows:

 0 = Butt (the default in xfig 2.1 and earlier)
 1 = Round
 2 = Projecting

    The arrow_type field is defined for LINES, ARCS and OPEN SPLINES
    only as follows:

 0 = Stick-type (the default in xfig 2.1 and earlier)
 1 = Closed triangle:
|\
|  \
|    \
|    /
```

```
|  /
|/
 2 = Closed with "indented" butt:
|\
\  \
 \   \
  \    \
  /    /
 /   /
/  /
|/
 3 = Closed with "pointed" butt:
   |\
  /   \
 /      \
/         \
\         /
 \      /
  \   /
   |/

    The arrow_style field is defined for LINES, ARCS and OPEN SPLINES
    only as follows:

 0 = Hollow (actually filled with white)
 1 = Filled with pen_color

(3.0) OBJECT DEFINITION:

    (3.1) Color Pseudo-objects (user-defined colors)
  This is used to define arbitrary colors beyond the 32 standard colors.
  The color objects must be defined before any other Fig objects.

    First line:
type name (brief description)
---------------------------
int object_code (always 0)
int color_number (color number, from 32-543 (512 total))
     hex string rgb values (hexadecimal string describing red,
 green and blue values (e.g. #330099) )

    (3.2) ARC

    First line:
type name (brief description)
---------------------------
int object_code (always 5)
int sub_type (0: pie-wedge (closed)
 1: open ended arc)
int line_style (enumeration type)
int line_thickness (1/80 inch)
int pen_color (enumeration type, pen color)
int fill_color (enumeration type, fill color)
int depth (enumeration type)
int pen_style (pen style, not used)
int area_fill (enumeration type, -1 = no fill)
float style_val (1/80 inch)
```

```
int cap_style (enumeration type)
int direction (0: clockwise, 1: counterclockwise)
int forward_arrow (0: no forward arrow, 1: on)
int backward_arrow (0: no forward arrow, 1: on)
float center_x, center_y (center of the arc)
int x1, y1 (Fig units, the 1st point the user entered)
int x2, y2 (Fig units, the 2nd point)
int x3, y3 (Fig units, the last point)

    Forward arrow line (Optional; absent if forward_arrow is 0):
type name (brief description)
---------------------------
int arrow_type (enumeration type)
int arrow_style (enumeration type)
float arrow_thickness (1/80 inch)
float  arrow_width (Fig units)
float arrow_height (Fig units)

    Backward arrow line (Optional; absent if backward_arrow is 0):
type name (brief description)
---------------------------
int arrow_type (enumeration type)
int arrow_style (enumeration type)
float arrow_thickness (1/80 inch)
float arrow_width (Fig units)
float arrow_height (Fig units)

    (3.3) COMPOUND

    A line with object code 6 signifies the start of a compound.
    There are four more numbers on this line which indicate the
    upper right corner and the lower left corner of the bounding
    box of this compound.  A line with object code -6 signifies
    the end of the compound.  Compound may be nested.

    First line:
type name (brief description)
---------------------------
int object_code (always 6)
int upperright_corner_x (Fig units)
int upperright_corner_y (Fig units)
int lowerleft_corner_x (Fig units)
int lowerleft_corner_y (Fig units)

    Subsequent lines:
objects
.
.

    Last line:
-6

    (3.4) ELLIPSE

    First line:
type name (brief description)
---------------------------
```

```
int object_code (always 1)
int sub_type (1: ellipse defined by radiuses
 2: ellipse defined by diameters
 3: circle defined by radius
 4: circle defined by diameter)
int line_style (enumeration type)
int thickness (1/80 inch)
int pen_color (enumeration type, pen color)
int fill_color (enumeration type, fill color)
int depth (enumeration type)
int pen_style (pen style, not used)
int area_fill (enumeration type, -1 = no fill)
float style_val (1/80 inch)
int direction (always 1)
float angle (radians, the angle of the x-axis)
int center_x, center_y (Fig units)
int radius_x, radius_y (Fig units)
int start_x, start_y (Fig units; the 1st point entered)
int end_x, end_y (Fig units; the last point entered)

    (3.5) POLYLINE

    First line:
type name (brief description)
---------------------------
int object_code (always 2)
int sub_type (1: polyline
 2: box
 3: polygon
 4: arc-box)
 5: imported-picture bounding-box)
int line_style (enumeration type)
int thickness (1/80 inch)
int pen_color (enumeration type, pen color)
int fill_color (enumeration type, fill color)
int depth (enumeration type)
int pen_style (pen style, not used)
int area_fill (enumeration type, -1 = no fill)
float style_val (1/80 inch)
int join_style (enumeration type)
int cap_style (enumeration type, only used for POLYLINE)
int radius (1/80 inch, radius of arc-boxes)
int forward_arrow (0: off, 1: on)
int backward_arrow (0: off, 1: on)
int npoints (number of points in line)

    Forward arrow line: same as ARC object

    Backward arrow line: same as ARC object

    Points line:
type name (brief description)
---------------------------
int x1, y1 (Fig units)
int x2, y2 (Fig units)
  .
  .
```

```
int xnpoints ynpoints (this will be the same as the 1st
point for polygon and box)

    PIC line:
type name (brief description)
---------------------------
boolean flipped orientation = normal (0) or flipped (1)
char file[] name of picture file to import

    (3.6) SPLINE

    First line:
type name (brief description)
---------------------------
int object_code (always 3)
int sub_type (0: open spline
 1: closed spline
 2: open interpolated spline
 3: closed interpolated spline)
int line_style (See the end of this section)
int thickness (1/80 inch)
int pen_color (enumeration type, pen color)
int fill_color (enumeration type, fill color)
int depth (enumeration type)
int pen_style (pen style, not used)
int area_fill (enumeration type, -1 = no fill)
float style_val (1/80 inch)
int cap_style (enumeration type, only used for open splines)
int forward_arrow (0: off, 1: on)
int backward_arrow (0: off, 1: on)
int npoints (number of control points in spline)

    Forward arrow line: same as ARC object

    Backward arrow line: same as ARC object

    Points line: same as POLYLINE object

    Control points line (absent if sub_type is 0 or 1):
Control points of interpolated spline.  There are two control
points for each knots.  A section i, of the spline is drawn
using Bezier cubic with the following four points:
(x ,y ), (rx ,ry ), (lx   , ly   ), (x   , y   ).
  i  i     i i      i+1    i+1     i+1   i+1
For closed interpolated spline the last pair of control points,
(lxnpoints,lynpoints) and (rxnpoints,rynpoints) (which can be ignored),
are the same as (lx1,ly1) and (rx1,ry1) respectively.

type name (brief description)
---------------------------
float lx1, ly1 (Fig units)
float rx1, ry1 (Fig units)
float lx2, ly2 (Fig units)
float rx2, ry2 (Fig units)
  .
  .
float lxnpoints, lynpoints (Fig units)
```

```
float rxnpoints, rynpoints (Fig units)

    (3.7) TEXT
type name (brief description)
---------------------------
int object  (always 4)
int sub_type (0: Left justified
 1: Center justified
 2: Right justified)
int color (enumeration type)
int depth (enumeration type)
int pen_style (enumeration , not used)
int font  (enumeration type)
float font_size  (font size in points)
float angle (radians, the angle of the text)
int font_flags (bit vector)
float height (Fig units)
float length (Fig units)
int x, y (Fig units, coordinate of the origin
 of the string.  If sub_type = 0, it is
 the lower left corner of the string.
 If sub_type = 1, it is the lower
 center.  Otherwise it is the lower
 right corner of the string.)
char string[] (ASCII characters; starts after a blank
 character following the last number and
 ends before the sequence '\001'.  This
 sequence is not part of the string.
 Characters above octal 177 are
 represented by \xxx where xxx is the
 octal value.  This permits Fig files to
 be edited with 7-bit editors and sent
 by e-mail without data loss.
 Note that the string may contain '\n'.)

    The font_flags field is defined as follows:

 Bit Description

  0 Rigid text (text doesn't scale when scaling compound objects)
  1 Special text (for LaTeX)
  2 PostScript font (otherwise LaTeX font is used)
  3 Hidden text

    The font field is defined as follows:

For font_flags bit 2 = 0 (LaTeX fonts):

 0 Default font
 1 Roman
 2 Bold
 3 Italic
 4 Sans Serif
 5 Typewriter

For font_flags bit 3 = 1 (PostScript fonts):
```

```
-1 Default font
 0 Times Roman
 1 Times Italic
 2 Times Bold
 3 Times Bold Italic
 4 AvantGarde Book
 5 AvantGarde Book Oblique
 6 AvantGarde Demi
 7 AvantGarde Demi Oblique
 8 Bookman Light
 9 Bookman Light Italic
10 Bookman Demi
11 Bookman Demi Italic
12 Courier
13 Courier Oblique
14 Courier Bold
15 Courier Bold Oblique
16 Helvetica
17 Helvetica Oblique
18 Helvetica Bold
19 Helvetica Bold Oblique
20 Helvetica Narrow
21 Helvetica Narrow Oblique
22 Helvetica Narrow Bold
23 Helvetica Narrow Bold Oblique
24 New Century Schoolbook Roman
25 New Century Schoolbook Italic
26 New Century Schoolbook Bold
27 New Century Schoolbook Bold Italic
28 Palatino Roman
29 Palatino Italic
30 Palatino Bold
31 Palatino Bold Italic
32 Symbol
33 Zapf Chancery Medium Italic
34 Zapf Dingbats
```

Index