Contents

The guide to CodeBuilder

Leonid E. Zakharov
CbResearch, Princeton NJ 08540

November 29, 2004

1 Introduction *(to ToC)*

As a software, the CodeBuilder system contains three parts:

1. `zcb` - the CodeBuilder itself, which is capable of recognizing a virtual Cb-structure intentionally inserted into some source files by formalized, comment-like statements. According to this structure standard control functionality will be provided to a final executable.

   Such an approach allows to organize the control of the code operation as well as communications of the code with the external World. It implements the idea of separation of data processing inside the code from its interaction with other codes, drivers, users and software.

   Based on interpretation of Cb-structure, `zcb` complements the original source code with a "Code Control" (`<CodeName>.Cb`) file and "Communication Control" files (`<SectionName>.CmC`), which can be used by a user for specifying the communications of each section of the code. Then, the "extended" source, which includes `*.Cb, *.CmC` files, is used by `zcb` for writing a new source code (on the same language as the original source). After conventional compilation, (and linking with Cb-library) this new source generates the executable having a flexible, standardized control of all sections and their communications.

2. `zhb` - the HelpBuilder, which generates the `.dvi` and `.ps` files for each Cb-section of the code and maintains their consistency with control parameters of the section and with formats of the I/O files.

3. `zdb` - the DocumentBuilder for organizing documentation for the code maintenance. While both `zcb` and `zhb` are serving the user in understanding and controlling the code, `zdb` is a helper for the code developer. It is based on the same structuring principles, now applied for organizing the global variables and routines inside the virtual Cb-structure of documentation.

   While all sections of communication Cb-structure are situated inside different routines, the documentation structure contains routines themselves and global variables as tokens. This technically separates the two Cb-structures in the source code.

   For each source file `zdb` can generate a separate document file having the same Cb-structure, where the routine code is represented by a skeleton, while a special space is provided for description of the routine functionality using LaTeX documenting capabilities.

   Such an approach of separation of the source and the doc-files with automatic maintenance of their consistency by `zdb` allows to keep the source code clean, with short names of variables and not contaminated by the comment lines. At the same time the best possible tools can be used for documenting routines of the code as well as the code as a logical structure.

   Separation of the source code and its documentation also allows safe distribution of the source code (thus, resolving platform consistency problems) without a fear that the author ideas of the algorithm would be intercepted by others.

The CodeBuilder is a non-intrusive software. All it requires is to insert (using the comment-like statements) the necessary information on structuring, which is absent in the conventional codes. Everything else is complimentary to the conventional source.

In practice, implementation of the contemporary control of the code and its communications does require some modifications of the original source code. The CodeBuilder reinforces thinking about the user and om transparent code control. In the case that these changes are not desirable, the software provides Insert/Delete capabilities. Additional (conventional) code can be written inside the `*.CmC` files (together with the label of the place in the source file) and then be actually inserted into intermediate source files when `zcb` generates a new source.

2 `zcb, zhb, zdb` technology *(to ToC)*

2.1 Installation *(to ToC)*

1. From `zcb.tgz` file, the CodeBuilder is expanded into `ZcbL/` directory by

   ```
   tar xzvf zcb.tgz
   ```

2. There is also a `qgd.tgz` with a code containing a Quick GuiDe into `zcb` which is expended into `Qgd/` directory by

   ```
   tar xzvf qgd.tgz
   ```

3. Then, the command

   ```
   ../ZcbL/mkcb
   ```

   called from `Qgd/` (or any other working directory at the same level) will generate executables `../ZcbL/cbL, ../ZcbL/hdL, ../ZcbL/dbL` and make them symbolically linked to `zcb, zhb, zdb` in the current directory. It also creates a few special files in the current directory.

4. The demo code is generated by the command

   ```
   Mkqgd
   ```

   It is useful only for introductory tutorial demonstration and is not a generic part of the system.

It is a convention that any working directory with codes, which use `zcb, zhb, zdb`, are situated at the same level as `ZcbL/`. This simplifies the access to Help, style files, libraries and other files of the Cb-system.

The call `../ZcbL/mkcb` *should be made at least once for each working directory of the codes, which use* `zcb`.

2.2 Working with `zcb` *(to ToC)*

.

When using `zcb`, instead of `make` the command

`Mk<CodeName>`

where `<CodeName>` is a given name to the code (3-4 letters), (re-)generates the `make` file and then, the executable `Cb<CodeName>`. As soon as `Mk<CodeName>` script was created, the work with the source files is the same as in conventional practice.

There are five stages of creation of `Mk<CodeName>` script, which should be passed through only once.

1. **Structuring the source of the code.** Starting from the file containing `int main()` routine, Cb-structure should be specified using special comment-like statements. This crucial step is explained in detail in Sect. 3.

2. **Creation of the Code Control file** `In/<CodeName>.Cb`**.** The simple command

   `zcb <MainFile>.c -i <CodeName>`

   generates a standard set of directories `Wrk In Out Doc Hlp Obj Csrc Fsrc` (if they were absent) and a file `Out/<CodeName>.Cb`, which contains the structural information about the code and its mapping to the structure of the I/O file system.

   After optional editing, the file should be moved into its final place `In/<CodeName>.Cb`.

   Option `-i` after `zcb` allows to specify the name of the code (use 3-4 letters for compactness). Without it the `<CodeName>` will be generated from the name of `<MainFile>.c` file.

3. **Creation of Online Help and Communication Control files**. The command

   `zcb <CodeName>.Cb`

   displays the future Control Panel of the Cb-code with its sections distributed in accordance with Cb-structuring. Three buttons, which will control the operation of the section, its Output and Input, are attached to each of Section on the CPanel.

   If the Control Panel seems to be erroneous, exit by `Cntr-c` and either restructure the source files or re-edit the `In/<CodeName>.Cb` file.

   If the Control Panel is correct, then `Esc` on keyboard, or clicking `<mouse-L>` on `Esc` button, creates (for every section) template files: `Hlp/<SectionName>.txt` (ASCII) and `Hlp/<SectionName>.Hlp` (LaTeX) for online help and `Out/<SectionName>.CmC` for communication control.

   Communication Control files `Out/<SectionName>.CmC` contain a number of sections (all disactivated by commenting them) for different kinds of communications. Their `Tbl_<SectionName>` section, which is responsible for interactive control of the section, contains all typical examples of syntax of description of communication objects.

   Edit `Out/*.CmC`-template files (see Sect. **??**) and move them into `In/` directory. These files can be prepared step by step. It is recommended to prepare, initially, only one `In/<SectionName>.Tbl` file. Other `In/*.CmC` files could be prepared gradually while working with the Cb-code.

   Only `In/*.CmC` *files, all of them, are used in generating the Cb-code.*

4. **Creation of** `<CodeName>.Src` **file with listing of source files**.

   Edit `Out/<CodeName>.Src` template file and substitute the names of source files and directories of the code. The structure of `Out/<CodeName>.Src` is self-explanatory. Move the edited file into a convenient place, e.g., into `Src/` directory.

   The `<CodeName>.Src` file is used by `zcb` for automatic generation of make files for Cb-code. This approach disengages the CodeBuilder from evolving syntax of `make` facilities, while allowing using capacities of `make` at full extent.

5. **Creation** `Mk<CodeName>` **code building command**. The call

   `zcb <CodeName>.Cb -m <CodeName>.Src`

   generates the make file `CbCodeName>.mk` and a simple script `Mk<CodeName>`. `Mk<CodeName>` *is a final product of five stages.*

Call
`Mk<CodeName>`
to generate the executable Cb-code `Cb<CodeName>` every time when changes were made in either original source files or in `In/*.CmC` files.

Only when the Cb-structure of the code was changed, repeat all the steps described above. Hiding existing Cb-blocks by commenting them in the `In/<CodeName>.Cb` file does not require repetition of these stages.

2.3 Working with executable Cb-code *(to ToC)*

Calling the resulting Cb-code

`Cb<CodeName>`

opens several X-windows (depending on the content of `In/*.CmC` files). The window "`Cb<CodeName> 0`" contains the Control Panel, while others are designated for controlling separate sections during the run of the code.

Clicking `<mouse-R>` on `<Help>` button in the CPanel window opens a simple ASCII help on how OnLine Help in its three forms (`.txt`, `.dvi` with hyperlinks, and `.ps`) can be accessed in Cb-code.

In a separate document, such as this one, is impossible to explain all details of controlling Cb-code and its communications. Because the system is evolving, it is a great burden to maintain documentation consistent with the state of software. Instead, the CodeBuilder provides the detail information on controlling the codes through the OnLine Helps system, where updates can be easily inserted.

Follow instructions available in the Help files.

3 Cb-structuring *(to ToC)*

Cb-structuring is an extension of conventional nested structuring based on Directory/File types of elements with a restriction that the order of elements does matter. Structuring should be understood before using the CodeBuilder. Other details of using CodeBuilder are typically prompted in the template or OnLine Help files.

Cb-structuring has three types of elements followed by a given name of the section

1. `blb_` is like a directory in the operational system and may include other structural elements.
2. `stg_` is like a file in the operational system and is the final element of the structure.
3. `mzl_` is a distributed entity, allowing to express possible virtual parallel processing in the code. Unlike `blb_` and `stg_` blocks, which are localized by their *begin* and *end* labels, each `mzl_` may be distributed along the entire code. It is fragmented by each `blb_` and may be intentionally fragmented inside `blb_` as well.

Topologically, `blb_<nameB>` may include only `mzl_<nameM>`'s, one or several (with possible repetition of names). Each `mzl_<nameM>` may include either `blb_<nameB>`'s or `stg_<nameS>`'s. Repetition of names of `blb_` and `stg_` is not allowed (although exceptions are possible). Also, `stg_<nameS>` cannot include elements of the same Cb-structure.

The skeleton of Cb-structure can be illustrated as

```
blb_B0{
  mzl_M0{
    stg_S00;
    stg_S01;
    stg_S02;
  }
  mzl_M1{
    stg_S10;
  }
  mzl_M2{
    stg_S20;
    blb_B20{
      mzl_M0{
        stg_S221;
      }
    }
```

```
    stg_S21;
  }
  mzl_M1{
    stg_S30;
  }
  mzl_M0{
    stg_S40;
    stg_S41;
  }
}
```

In some sense, Cb-structuring compensates a deficiency of conventional "Dir/File" structuring and makes it better fitting to the people style of thinking. It is also more consistent with the needs of parallel programming (if `mzl_`'s are associated with virtual processors) and understanding interactions inside the complicated numerical codes.

A simple illustration could be an abstract criminal/detective story. While the table of contents (based on Dir/File-like sections) of the book conveys the structure of the story as the time development, there is no possibility to trace the development associated with either criminal or detective themselves and their interaction. Cb-structuring would allow this by specifying `mzl_Criminal` and `mzl_Detective` in the relevant `blb_` sections.

Depending on the context, different interpretations and properties can be given to Cb-elements. E.g., `zcb` maps some of `blb_` into the structure of the I/O file system and makes them repeatable during the code run, while `zdb` analyzes consistency of the use of global variables by different routines with their position in the Cb-structure and reflects the results of analysis in the document files.

Essentially, Cb-structuring is a first principle approach based on irreducible set of structural elements. Upon implementation of its structure recognition routines and different drivers for its structural elements it is applicable for many purposes in organizing control of numerical codes, generating documentation and information processing, all made in a mutually consistent manner. Three codes `zcb, zhb, zdb` are just some particular implementations of interpretation of Cb-structuring.

4 zcb-Code Builder *(to ToC)*

This section explains the basic syntax associated with use of `zcb` and its illustration with a demonstration code `Cbqgd`. As soon as the basic idea of the syntax is understood, the use of `zcb` becomes simple because all the characteristic examples are present in the template files automatically generated by `zcb`.

4.1 Demo code on using zcb *(to ToC)*

The directory `Qgd/` contains a demo code `Cbqgd` which demonstrates the code building process using a simple set of examples. (Its source is `Src/qgd.c` and the source list file is `Src/qgd.Src`.) Run it by calling

`Cbqgd`

Then, hit `Esc` and follow the instructions in its window `qgd 1`.

4.2 Working directory for Cb-code *(to ToC)*

Any directory at the level of `../ZcbL/` can be a working directory for Cb-code generated by `zcb` from original conventional source. This directory will be the reference point for different service directories generated and used by `zcb`.

4.3 Syntax of Cb-structuring for `zcb` and of `.Cb` file *(to ToC)*

4.3.1 Structuring C-codes *(to ToC)*

Cb-structure accepted by `zcb` starts inside the `int main(...)` routine and can be extended to other routines and files. It is implemented with pairs of C-preprocessor directives `#ifndef, #endif` specifying extent of each Cb-section, like in the example taken from ESC (Equilibrium and Stability Code)

```
int main(int argc, char **argv)
{
#ifndef mzl_ESC
    ...;
#ifndef blb_Machine
#ifndef mzl_ESC
#ifndef stg_PlLim
    ...;
#endif
#ifndef stg_MFProbe
#endif
#ifndef stg_PFBlocks
#endif
    ...;
#ifndef blb_Year
#ifndef mzl_ESC
    ...;
#ifndef stg_MSE_Geom
#endif
    ...;
#ifndef blb_Shot
#ifndef mzl_ESC
        ...;
#ifndef stg_WFormB
#endif
#ifndef stg_WFormI
#endif
        ...;
#ifndef blb_Equil
#ifndef mzl_ESC
          ...;
#ifndef stg_PlVac
            ...;
#endif
#ifndef stg_PFCIeq
            ...;
#endif
#ifndef stg_PlPr
#endif
#ifndef stg_PlCr
#endif
          ...;
  do{
              ...;
#ifndef stg_InsFSol
#endif
            ...;
  }while(...);
#ifndef stg_EqOut
#endif
#ifndef stg_BalSt
#endif
#ifndef stg_Orbits
#endif
          ...;
#endif
#endif/*blb_Equil*/
        ...;
#endif
#endif/*blb_Shot*/
      ...;
```

```
#endif
#endif/*blb_Year*/
  ...;
#endif/*mzl_ESC*/
#endif/*blb_Machine*/
  ...;
#endif/*mzl_ESC*/
  return(0);
}
```

This struture has 4 nested `blb_` blocks (`Machine, Year, Shot, Equil`) which contains a number of `stg_`'s. Comments after `#endif` does not matter for `zcb`.

Ugly looking, preprocessor directives are easy to trace by the editors and they cannot be removed if preprocessing is involved in generating the C-source files from the higher level storages (like Khuth/Krommes WEB/FWEB).

There are following rules in structuring for `zcb`:

1. Structure starts with `mzl_`, which topologically encloses all other section. In Cb-code all of it will be inserted into `blb_<CodeName>` lock, which makes the structure closed.

2. *Temporarily, only one* `mzl_` *name can be present in the structure.*

3. Boundaries of Cb-sections should not violate the topology of C-language blocks in the code. In Cb-code, it eventually will be converted into actual C-language blocks. At present, `zcb` does not check the consistency of Cb-structure with original C-blocks.

4. It should be no jumps into or from Cb-sections without interseting their boundaries. Intersection of boundaries of Cb-structure are used by `zcb` library to trace the run for controlling Cb-code.

5. Otherwise, Cb-structure directives can be inserted at any place in the code, where run interseption, intervention, control, I/O or communication are necessary. Even a section with no code lines inside may have a sense for controlling the code and communicating with external devices.

   E.g., `zcb` library provides the possibility of automatic activation of interactive regime in the first Cb-section having it, when some condition was met. This would stop the code and allow some interactive steering in the abnormal situation without crushing the run.

6. Names of Cb-sections are used for generating paths, directory names and file names and, thus, characters which impede name interpretation by operational system should be avoided. For `zcb` only `'('`, `')'`, `'{'`, `'}'`, `'/'`, `';'` are rigorously prohibited in names.

7. Structuring can continue in another source file by referencing it, e.g., as in example from inside the main file

   ```
     ...;
   int main(int argc, char **argv)
   {
     ...;
     ...;
   #ifndef blb_B0(F=Src/fname1.c)
     ...;
   #ifndef stg_S5
   #endif
     ...;
   #endif/*blb_B0*/
     ...;
   ```

   Inside the Src/fname1.c file it should continue, e.g., as

   ```
     ...;
   ```

```
int Routine0(int K)
{
  ...;
  ...;
#ifndef blb_B0
  ...;
#ifndef stg_F0
#endif
#ifndef stg_F1
#endif
  ...;
#endif/*blb_B0*/
  ...;
  return(0);
}
```

When `zcb` encounter the reference directive `(F=Src/fname1.c)`, it goes to the file `Src/fname1.c` and searches for `#ifndef blb_B0` and parces the file till the corresponding `#endif` and then returns to the point of reference. Nestedness of references is arbitrary.

At this moment only `blb_` *may carry a reference.*

4.3.2 Incorporation of the Code Control into C-source *(to ToC)*

At Cb-code building stage (by invoking `Mk<CodeName>`) when `zcb` generates a new source file `Csrc/Cb<FileName>.c`, it makes insertions right after `#ifndef blb_` and `#ifndef stg_` lines and before corresponding `#endif`'s, e.g.,

```
  ...;
#ifndef blb_B0
  while(CbBlbModeOff(3) == 0){
  ...;
#ifndef stg_F1
  while(CbStgModeOff(3,0,5) == 0){
  ...;
    if(CbIfExitStg()) break;
  }
#endif
  ...;
    if(CbIfExitBlb()) break;
  }
#endif/*blb_B0*/
  ...;
  return(0);
}
```

`zcb` library routines `CbBlbModeOff(...), CbStgModeOff(...), CbIfExitStg(), CbIfExitBlb()` are controlled by `Mode`-switches on the Control Panel of Cb-code.

4.3.3 Specifying the structure of I/O file system using `.Cb` file *(to ToC)*

Based on interpretation of Cb-structure specified with `#ifndef, #endif` pairs, `zcb` creates the Code Control file `<CodeName>.Cb`, which contain the structure in a compact form:

1. Each `blb_` inside `<CodeName>.Cb` has a member `OD` (standing for output directory). If it is not empty (name does not matter), e.g.,

   ```
   OD=Year;
   ```

   then a name of directory will be attached to this `blb_` in the Control Panel of Cb-code. The actual name of directory can be introduced before launching the run of Cb-code, and all sections inside this

blb_ will write into this directory. Empty member, i.e., (OD=;) means that it will be no directory corresponding to blb_.

Editing OD *inside* <CodeName>.Cb *creates correspondence between the structure of Cb-code and its I/O file system.*

As soon as <CodeName>.Cb is edited and moved into In/ directory parsing of all the source files is not necessary. After restructuring the source files and running zcb on them, the information from existing In/<CodeName>.Cb will be transfered into the new Out/<CodeName>.Cb.

4.3.4 Reinforcing consistency in switches on Control Panel *(to ToC)*

By default, the combinations of switches on Control Panel responsible for mode of operation and I/O of each section, are arbitrary. For example, some section cannot be turn Off, if some others remain active.

A syntax of a mechanism in <CodeName>.Cb file, which would restrict combinations of switches, is under design.

4.4 Communication Control files and their syntax *(to ToC)*

The Out/<SectionName>.CmC templates are generated by zcb automatically as describes in Sect. 2.2. They contain a full set of sections which can be used for arranging communications and interactive control using zcb capabilities.

The syntax of each section is the following

<Type><SectionName>(...){
  description of communication objects
}

Anything between sections (except C-like commenting /* ... */) is ignored by zcb.

Parentheses (...) contain parameters of the section, separated by ','

Parameter ID	Section	Opt	Comment
W=2	Tbl_	Obligatory	Window used by section for interactive regime
SF="Src/aaa.c"	*All*	Obligatory	Name of the source file where comunication is localized
L=4	Tbl_	Optional	Number of lines in the GUI Table
Beg="labelB"	*All*	Obligatory	Image for begin of the scope of communication
End="labelE"	*All*	Obligatory	Image for End of the scope of communication

Templates intend to contain a comprehensive set examples of the syntax and its explanation (in Hlp section of the files). *The basic syntax of communication objects is the same for all sections of* *.CmC *files.*

The templates for blb_ and stg_ blocks are different, although the syntax is the same (but will be a little bit different in future to express a special role of blb_). *Use* stg_ *templates as a guide for* blb_ *blocks as well.*

While generating new source, zcb takes into account all In/*.CmC files in the In/ directory.

4.4.1 Scope of communication sections of In/*.CmC files *(to ToC)*

By default, zcb should include C-instructions into the code, which would probide the scope of each section extending to the entire section.

!! It is not implemented yet in this way !! Instead, it includes these instructions right after the image specified by Beg= *and before the image specified by* End= *in parameter list.*

The presence of these instructions does not affect the original code, if the communiction is desactivated by a CPanel switch (or if at the moment the code is in another Cb-section, than the section communication belong to. Otherwise, by entering the communication block the corresponding file (or channel) will be open. It will be closed at the end of the block.

The block for interactive regime (Tbl_) can be repeated by hitting <Enter> (or using mouse click on button <Enter>) and exited by hitting <Esc> (See, the OnLine Help in this regard while running Cb-code).

Templates contains self-consitent images for labeling the scope of communication sections. It is recommended to use them because they are easily recognizable by the document builder `zdb`.

The pointwise communication needs only `Beg=` specification.

Follow examples in templates to describe objects specific for each section of a code. After moving step by step `Out/*.CmC` *files into* `In/` *dierectory and creation of executable Cb-code use OnLine* `<Help>` *for practicing.*

The following sections describe the syntax of typical sections of `*.CmC` files in a formalized way.

4.4.2 `Tbl_` section of Interactive regime *(to ToC)*

4.4.3 `Hlp_` section for control parameters *(to ToC)*

4.4.4 `Plt_` section for graphics of Interactive regime *(to ToC)*

4.4.5 `AscI_` ASCII I/O sections *(to ToC)*

4.4.6 `BnrR_` binary I/O sections *(to ToC)*

4.4.7 `Include/Delete` mechanism *(to ToC)*

4.5 Syntax of `<CodeName>.Src` source list file *(to ToC)*

The file contains a structured list of source files of the original code, as well as instructions for possible drivers. At this moment only a section, specifying generation of make-file is present.

`zcb` *does not generate a template for this file. This will be corrected soon.*

4.5.1 Structure of `<CodeName>.Src` file *(to ToC)*

A simple example (for `bst` code) is

```
bst{
  .mk[
      CC =cc -c;
      LINK =cc;
      include FLIB.inc;
        *.o(.c) :  $(CC);
      bst(*.o) :$(LINK) -o bst *.o \
      $(FLIB) -lGLU -lGL -lXmu -lm;
  ];
  Src/{
    bst.c[.o];
    bstD.c(bst.h)[.o];
    bstA.c(bst.h)[.o];
    bstN.c(bst.h)[.o+=-DDEBUG];
    bstB.c(bst.h)[.o];
    bstG.c(bst.h)[.o];
    bstGL.c(bst.h)[.o];
  }
  ../Esc/Src/{
    esiZ.c[.o];
    splines.c[.o];
  }
  ../ZcbL/Src/{
    cbGL.c[.o];
  }
  ../Lib/Src/{
    NumRec.c[.o];
  }
  /u/wrk/3D/Src/(3dH.h)[.o]{
    3dT.c;
    3dP.c;
    3dO.c;
```

}
}

The file simply reflects the structure of the source code file system inserted into `<CodeName>{ ... }` block. Sections `.<DriverExtension>[...]` are situated before the source file structure.

!! Do not forget to put a proper `<CodeName>` *when generating this file by copying it from another code !!.*

The name of files are followed by `(...)` specifying dependence on other files. Their paths are calculated relative to the source file directory.

The content of next `[...]` brackets specifies possible extensions (separated by `';'`) which different drivers can generate from the file. Each option can include `+=...` or `-=...` as additional (or excluded options) options to those specified separately in driver section.

If content of `(...)`, `[...]` is common for all files of directory, it could be put after its name and droped from file lines (like in the last block of the example).

4.5.2 Make section of `<CodeName>.Src` *(to ToC)*

The section `.mk[...]` contains instructions for generating the make file of the (original) code. It mimics the most primitive conventions of conventional `make` syntax, i.e.,

1. `'='` is used to make definitions, while `$(...)` to substitute definition.
2. `*.o(.c) : $(CC);` tells that all `*.o(.c)` files depends on corresponding `.c` files. `':'` separator is followed by instruction applied (implicitly) to `.c` file. If description of file contains specific instruction, then they be included for this file at the end of the common instruction line.
3. `bst(*.o) : $(LINK) -o bst *.o ...;` tells that `bst` object (executable) should be created according the instruction, where `*.o` will be substituted by all output of instructions for `*.o`.

All other "bird language" of conventional `make` *files is droped as unnecessary.*

4.6 Basic functionalities of `zcb` *(to ToC)*

At the stage of code building `zcb` functions are:

1. To recognize the communication control structure of the code and to generate the template files for the code control (`In/<CodeName>.Cb`), its communications (`In/<SectionName>.CmC`) and online help (`Hlp/<SectionName>.txt` for ASCII and `Hlp/<SectionName>.Hlp` for LaTeX).
2. To generate new source files (`Csrc/Cb<SourceFile>.c`) based on original (`<SourceFile>.c`) source files and `In/<CodeName>.Cb`, `In/*.CmC` files.
3. To generate a make command `Mk<CodeName>` which regenerates `Csrc/Cb*.c` files and makes an executable `Cb<CodeName>`.

In addition to already existing properties of the original code, the executable `Cb<CodeName>` has the following functionality

1. Uniform and distributed control of the code itself and of each of its sections.
2. Interactive control of the code sections.
3. Access to the OnLine Help (`*.txt, *.dvi, *.ps` for each section and for each control parameter of the section.
4. Organized Input/Output of the code into the automatically generted file system. Its structure is consistent with Cb-structure of the code and its I/O files are named by Cb-sections generating these files.
5. Organized internal structure of both ASCII and binary I/O files.