CRiSP File Editor

Programmers Guide

Version 6

www.pacemaker.co.uk

sales@pacemaker.co.uk

fox@crisp.demon.co.uk

.

Introduction

Document version: 1.002, date 24 November 1997

This guide discusses how to write macros in the CRiSP extension language, known as **CRUNCH**. The CRiSP software package is a highly flexible file editor for use by people who need to edit files, whether they be programmers, or engineers. CRiSP is designed to be easy to use and have a familiar user interface. Underlying all this power is a powerful macro language which implements the things you use and see in the user interface.

You may want to write your own personal macros for CRiSP in order to make private tweaks to keyboard bindings, or to write your own subsystems to do more complex actions. The CRiSP binary itself is a program interpreter, interpreting the steps in the macros in order to perform some well defined action. As an interpreter it has a lot of power and allows such things as keyboard mappings to be set up, editing of files, creating dialog boxes, configuring colors, etc. Just about everything you see within the CRiSP user interface is built up from macros, and there is therefore a lot of expressive power in the software.

This programming guide covers the following areas:

→ Getting started with macro programming (pg. 7).

→ General introduction to macro programming(pg. 8).

→ CRUNCH compiler,.(pg. 14).

→ CRUNCH language,.(pg. 14).

→ Macro debugging(pg. 30).

→ High-level data objects (e.g. windows, buffers)(pg. 38).

→ How to create custom colorization languages(pg. 67).

→ Interprocess communication(pg. 79).

→ Description of the Dialog Box subsystem.(pg. 51).

→ Macro Primitives Guide

Disclaimer

This manual and the CRiSP Macro Primitives manual describe how to use the facilities in CRiSP for extending its facilities. Foxtrot Systems Ltd makes no guarantee on the validity of the information contained herein. In addition, any errors in the macro language (compiler or CRiSP) may or may not be fixed at a later date. Although the facilities described have been extensively tested, the language and tools have been designed to provide an efficient and portable user interface to the editor. Any private use of these facilities which lead to core dumps, crashing of machines, or loss of files will not be construed as a deficiency in the software supplied.

Where possible, errors leading to core dumps or machine crashes will be fixed if a suitable bug report form (see User Guide for a copy) is enclosed. Due to the nature of programming it is not possible to guarantee that every combination of primitives will lead to expected behaviour.

Getting started with Macro programming

This section is designed to introduce the basic things you need to know to write your own macro. If you require further information, consult the introduction at the top of the **Programmers Guide** on-line help.

If you have never programmed a macro in CRiSP before then here are some tips to get you started:

The macro language (known as **crunch**) is an ANSI-C like macro language, e.g. you define functions which are callable (from the Command: prompt, for example), and which can be assigned to keys. The macro language provides numerous data types and internal builtin functions. The most common data types are strings and integers. (crunch does not provide support for pointers, but you will find you do not need these anyhow).

Macro files end in the file extension **.cr**. To create a macro file, edit a new file with the appropriate extension. For example, create a macro file called **mymacro.cr**. Insert the following into the buffer:

```
# include   <crisp.h>
void
mymacro()
{
      message("This is my macro!");
}
```

The #include statement is something you will need sooner or later in the macros you write. Although it is not needed for this simple macro, you may need it as you extend your macro. The **crisp.h** file contains numerous constant definitions which you need for some of the builtin functions of CRiSP.

This macro file contains a single macro, called **mymacro.** You can execute this macro after you have compiled and loaded the macro. On most machines, just press the key **<Alt-F10>** and the macro file will be compiled and loaded into memory.

On some Unix platforms running the Motif window manager, you may find that **<Alt-F10>** zooms the CRiSP editing window to the full screen size. In this case, execute the command **load** at the **Command:** prompt. You can access the Command: prompt by pressing the **<F10>** key.

After compilation, you will find a file called **mymacro.cm** in the same directory as the original source. This is the file which is important to CRiSP. (On the other hand, the file **mymacro.cr** is important to you).

You can add multiple macros to your macro file and build up complex personal macros to do whatever you want. It is a good idea if you are going to do extensive macro writing to browse the CRiSP supplied macros and investigate whether some function you want to write is not already available directly, or callable as a subroutine.

If you want to have your macro file loaded automatically on startup then enable the **Options**→**Startup**→**Startup macro** menu option. Type in the full path to your **.cm** or **.cr** file.

An Overview of the Programming Facilities

CRiSP is an interpretive language execution engine, combined with support for very high level data objects. The types of objects CRiSP knows how to manipulate is much more than for a standard programming language. CRiSP supports basic primitive data types, such as integers, floating point numbers and strings, as well as high level data objects such as buffers, dialog boxes, keyboard mappings, etc.

The function of CRiSP as a file editor is the combination of the execution engine (the CRiSP binary) and the various macros supplied as part of the distribution. The supplied macros provide a wide variety of user interface functions which can be tailored, by virtue of the various set up options, or reprogrammed by changing the macro sources.

The macros are supplied in a source form, which is a language loosely based on the ANSI-C language, and a compiled format. The source language provides various programming facilities and is designed so that macro programmers can write and document maintainable macros. Macros can be very sophisticated and involved, so macro programmers should create and support macros with respect. It is possible to write powerful one-liner macros, but if you are going to be writing a lot of macros, then you will need to organise things so that you can review and update the macros at a later date. This is no different from programming in any ordinary language.

The compiled format is designed to be loaded into CRiSP much faster than raw interpretation of the macro source can be. But this does mean that you have to actually compile the macro sources before CRiSP can execute them. Fortunately this is very easy and CRiSP provides various facilities to help in this.

The underlying *machine-code* of CRiSP is a Lisp-like language. The CRUNCH language is compiled into an internal form of the lisp language. The Lisp-like language has no official name, but files written using this syntax normally have a '**.m**' file extension. Crunch language files have a **.cr** file extension. Compiled macros have a **.cm** extension.

The lisp language may be considered the *assembly* language of CRiSP. It is exceptionally rare to code directly in the lisp language since the CRUNCH language provides a superset of functionality (such as consistency checking, common expression elimination, and other optimisations), plus the code is much more maintainable.

See also

→Getting started with macro programming (pg. 7).

→Writing your own macros(pg. 9).

→The .m language(pg. 90).

→The crunch language(pg. 14).

Writing your own macros

Private macros can be loosely categorized into two types: small one-off macros which are used to make some piece of functionality easier to use (e.g. mapping certain editing commands to special keystrokes), or major projects in their own right. The entirety of the CRiSP product can be broken down into major subsystems (Unix mail, the setup dialog boxes) or simple value-added services, such as capitalizing words, repeating the last search.

The bits that go to make up CRiSP as a whole are structured in a way that has allowed CRiSP to evolve and allow room for manoeuvre as future functionality is added. CRiSP and the macro environment is a bit like the Microsoft Windows 3.x environment in that all the macros are co-operative. Most macros are independent of one another but within the context of a file editor, the macros are all adding value to the software.

When you come to write your own macros, you will need to think and understand what you are trying to achieve. For one-liner macros, there is little to think about and you can pretty much achieve what you want once you understand the mechanics of CRiSP. If you are attempting to build a complex macro then it can be useful to step back and think about what you want to achieve. Much of this is common sense and applies to software development in general, but it is worth understanding the environment you are going to be programming in. Much of this manual is devoted to explaining the technical concepts and issues in writing your macros. This section is more concerned with taking a steady pass over the concept of macro writing.

There is nothing magic or special in writing macros to extend CRiSP for your own personal desire. There is a great sense of achievement in mechanizing some tedious editor task which has frustrated you in the past. So in a sense, customizing the editor can have its appeal.

One of the major goals of the CRiSP macros is an attempt to achieve an object-oriented structured design. (The term 'object-oriented' is used in a loose sense). When writing a macro to achieve something it is desirable to ensure that the macro you have does not interfere with any existing macro. Also, looking to the future, you need to ensure that other newer macros cannot affect the functionality of your macros. For a normal high-level programming language, once you have written a program, debugged and compiled it, that is it. Nothing can affect the correct behaviour of that program.

With an interpretive environment like CRiSP you are not in control. Rather, it is like being a guest in someone elses house - you have to obey the rules or else you will get into a muddle. If we consider that the environment of CRiSP, the macros and the binary, is one giant program, then in effect what you are doing when writing your own macro is customizing the existing behaviour - you are interfering with the existing code. The problems that can strike firstly is accidentally reusing symbol or function names which some other macro is using, or creating private system buffers or files which some other macro also uses. Most of the time when this happens, it isn't that difficult to figure out what is going wrong, but it is a nuisance that you cannot code something and prove it is correct without considering the rest of the system.

When writing large C or C++ programs, split into multiple source files, you use programming conventions to avoid such things as name space pollution, e.g. use of statically scoped variables and functions. CRiSP encourages you to do likewise. Something that is declared **static** cannot be accidentally affected by some other macro and gives you a sense of protection. Scoping and global vs. static are discussed in more detail in the section on "Macros".

Another thing to consider before resorting to a private macro is to understand CRiSP, as a user, and its philosophy. CRiSP is a complex piece of software when taken as a whole, yet it strives to create an easy to use user interface for non-technically oriented people. Many people will use certain aspects of the software and ignore or be totally ignorant of other aspects. What this means is that the functionality you are after may already be there. If you have work to do and really find something annoying in CRiSP then by all means, go ahead, and create your own personal macro(s). If you are going to spend a lot of time macro programming then it is worth examining CRiSP as a whole because you may find useful code libraries or new ideas on how to achieve things in a faster or more flexible way than any of your original ideas.

There are three things that are worth bearing in mind before embarking on writing your first macro:

1. Learn to use CRiSP properly. Do not be afraid to try things out, as otherwise you may not understand what the supplied macros are actually doing.
2. Look at the sources to the macros which come with the CRiSP distribution. These not only implement

the usability features of CRiSP but also contain useful real-life examples of various aspects of CRiSP macro programming.

3. Where possibly, try and maintain a macro programming style. Do not write *throwaway* macros - treat them with respect. For example, format them nicely, comment them, lay them out well. You will find it so much easier to revisit your macros at a later date, or if you need to pass them on to people, they will actually understand your code. Of course, this is what you should be doing with all your code, whether it be CRiSP or C or Ada.

The macros supplied with CRiSP cover a lot of ground. These macros have evolved and grown over the years. Some of these macros represent good solid examples of programming, others are not quite in the same calibre. Many of these macros have evolved from an experiment to real use. So the coding styles are not necessarily consistent. As each new version of CRiSP is released, new ideas are tried and the macros are refined. As an example the capability to declare static variables and static functions is a relatively recent addition to CRiSP. So not all functions which should be are marked with the *static* storage class specifier.

If you are planning to write a large complex macro, then the best recommendation is to start off with something simple - experimenting at each step of the way, slowly refining and adding the required functionality. This is how all of the functions in CRiSP have evolved. If you write a large macro without experimentation, then you may find it very difficult to debug unless you have a good understanding of all the information available. Although CRiSP looks and feels like a C interpreter, you should remember that it is really an extensible file editor. By being very focused in your aims and expectations, you can know what to look for.

Tools Overview

This section provides command line summary information of the various command line tools provided with CRiSP. As a macro programmer, the tool you are most likely to use is the **crunch** compiler. If you are not interested in writing CRiSP macros, then you are more likely to find the **crtags** tool of use to you.

cm(pg. 90).	This is a low level macro language compiler. It is provided for completeness and backwards compatibility with the older BRIEF **.m** style macros.
crpp	This is the macro language preprocessor. It is very similar to a C or C++ preprocessor, but is supplied as part of the macro compilation system, since it is not possible to rely on customers having a C compiler installed on their system. It is not normally invoked directly, but is used by the **crunch** compiler.
crunch(pg. 14).	This is the main macro compiler which you can use to compile source files in the crunch language (extension **.cr)** into compiled macro files (extension **.cm**). This is invoked automatically if you use the **<Alt-F10>** key in CRiSP to compile the current buffer, or the **load** command at the **Command:** prompt.`<Alt-F10>`
crtags(pg. 10).	This command is used to generate a cross reference tags file for source files. A number of languages are supported. Although this command has nothing to do with macro writing (it can be used on crunch macro files however), it is described in this document. CRiSP provides a full user interface for manipulating tag files - creating tags, and using the class browser window to view the objects defined in a user's project.

crtags: cross-referencing source files

`crtags`The **crtags** program is an enhanced tags utility loosely based on the Unix program 'tags' which is normally used in conjunction with the 'vi' editor. A tags file is a file which contains a database of all language specific constructs of source files, e.g. function definitions, type definitions, constants, etc.

The crtags program can be used to scan source files in various languages and produce a database file listing the occurrences of important elements of the language, such as function definitions, constant definitions, structures, classes, etc.

The supported languages are listed later in this section together with some application notes relating to each language.

You can get a quick command line summary using the '-help' command line switch.

The basic command line syntax is:

```
      crtags [switches] file1 file2 ....
```

You can use Unix style wild cards for the filenames, even on the Windows platforms.

The command line switches can be grouped into three types:

- general purpose switches affecting the crtags program,
- output file format switches
- language sensitive element switches

File formats

crtags creates a tags file. Two file formats are supported: text and binary. The binary format is smaller and contains more information needed by CRiSP's cross-referencing facility (the class browser window). The text file format was used in older versions of CRiSP and has now been deprecated.`File formats:crtags`

The binary file format is designed to be machine independent, meaning that if you share a network filesystem, then the tags file can be used by all CRiSP clients on the network no matter what CPU architecture they use.

Specifying Directories

If you specify the name of a directory instead of a file then crtags will recursively scan that directory for files. This can be a quick and convenient way of handling entire projects of source code.

Language Mapping switches

crtags allows you to specify language mapping switches on the command line. This is designed to allow you to configure how files with non-standard extensions are handled. By default, crtags treats files with certain extensions as detailed below.

To set the language mapping mode, you can put something like this:

```
      xyz=pascal
```

on the command line. Multiple switches can be used. The left hand side should match in case the extensions to use. The right hand side should be one of the standard file extensions recognized or the language name.

General switches

In general, it is recommended to use crtags without any command lines flags - the default options are sufficient to create a browser tags file.

-a	Appends the new tags information to the end of the specified tags file rather than replacing it.
-all	When parsing files, duplicate definitions are normally ignored. For example in a C source file, you might have a prototype for a function and the function definition itself. If you specify this flag then all occurrences of the object will be listed in the tags file.
-absolute	If this switch is specified then the filenames in the tags file will be the full path to the file. This can be useful when you have a large project which spans many directories and you want to jump to arbitrary functions in any directory. The down-side of this switch is that the tags file may be significantly larger than if the abbreviated filenames are used. This is the default.
-binary	Create binary format tags file (default). The binary format allows for faster cross-referencing in the CRiSP browser and supports structural information needed to display the class browser window.
-d	Enables debug. Not useful for the end user.
-help	Lists complete summary of supported languages, switches and options.
-I file	This switch allows you to specify the name of a file containing a list of filenames to be

	included on the command line. This allows you to generate a list of files and store them in a file rather than being exposed to the command line limitations of certain operating systems. Multiple -I switches can be specified on the command line and can be intermingled with normal files. If you use this switch is must be the last switch on the command line.
-ignorecase	When sorting the tags file during the output stage, ignore the case of data elements.
-len nnn	Specifies the length of the context line to include in the tags file. The default is 10, which means that for each tag, not only is the line number within a file recorded but also a portion of the matching line. This is designed to be used, because as files are edited, the recorded line number may no longer exactly match the recorded tag. In this case, the CRiSP tag macro will search for the line using the actual contents of the line. Large values will significantly increase the size of the tags file, so there is a trade off between speed and file size.
-l <lang>	Treat all files on the command line as if they were of the specified language type. The complete list of available languages can be seen using the '-help' switch, or consult the sections below.
-nologo	Do not print out the copyright logo message.
-O	Optimise the tags output file. This only has meaning when used with the 'crtags' file format and reduces the size of the tags file at the expense of readability.
-o <tags>	Specifies the name of the output file to receive the tags database. The default value is "tags" in the current directory.
-q	Quiet mode. Do not display progress messages as files are parsed.
-sort	Turns off the sorting of functions in the tags file.
-regexp <re>	
+regexp <re>	These two switches allow you to control the tag entries which are placed in the database. If the **-regexp** switch is used then all entries which match the expression are **not** placed in the file. If you use the **+regexp** switch then only entries which do match are placed in the database.`regexp:crtags switches`
-text	Create an ASCII (old style) index file. The text file is in a human readable file format, but is not as efficient as the default binary format, and in addition it does not support information needed for the CRiSP browser.
-u	Update mode. Not currently implemented.
-w	Enables warnings. Used for compatibility with Unix *'ctags'* to show multiple function definitions. Not particularly useful for larger projects where duplicate static functions may exist in multiple source files, or where conditional compilation can cause two definitions for the same function to be recorded.
-x	Create cxref style output. Not currently implemented.

Filename filtering

When performing recursive directory scanning, you may want to skip certain files or directories, for example those used to store source code archives.

-xd <dirname>	Specify directories to be skipped.
-xf <filename>	Specify filenames to be ignored.

These switches can be specified multiple times as needed. When specifying directory names, only specify the last component of the directory to be skipped, e.g. **-xd SCCS** to ignore all SCCS subdirectories.

Filename and directory specifications can include shell wild cards, such as *, [..] and ?. When specifying these on a command line, you may need to quote the argument to avoid expansion interactions with the command line shell you are using.

Output file format

These switches allow you to specify the output file format.

-tags	Output is compatible with the Unix ctags program and hence the tags database can be used with other editors such as vi.
-text	Output file in textual format. Useful for diagnosing the output.
-crtags	(Default) Proprietary file format designed to be used in conjunction with CRiSP. Although this file format is liable to change in the future, it provides more information in the tags file which at some point in the future will be used by CRiSP to provide a more user-friendly interface.

Language specifications"

crtags contains a table of default file extensions and the languages they correspond to. If you need to override these language formats, then use the '-l <lang>' switch to specify the language to be applied to ALL files on the command line. (This switch is not position sensitive, it must precede all filenames on the command line).

Language element feature selection

crtags supports a variety of languages. Different languages can describe different structures, e.g. a C++ program can contain class definitions, whereas an Assembly language program cannot. The data elements which are currently parsed by crtags are listed below for each of the available languages.

You can use the -FEATURE switch to disable tag generation for some of these language features. For example, you may not want #define entries in the tags file.

The following lists the elements you can disable, although not all of them are available for every language - see the per-language description below for a list of entities supported.

-CLASS	Discard class definitions
-CONST	Discard constant variable definitions (or parameter definitions)
-DATA	Discard definitions of global variables.
-DEFINE	Discard #define definitions.
-ENUM	Discard enum definitions
-ENUM_MEMBER	Discard the members of an enum definition.
-FUNCTION	Discard function definitions
-INDEX	Discard SQL index definitions.
-LABEL	Discard label definitions (targets of a goto statement).
-MEMBER	Discard structure/union/class member definitions
-MODULE	Discard module definitions.
-PACKAGE	Discard package definitions.
-PROCEDURE	Discard procedure (subroutine) definitions
-RULE	Discard SQL rule definitions.
-STRUCT	Discard C style structure definitions
-TABLE	Discard SQL table definitions.
-TRIGGER	Discard SQL trigger definitions.
-TYPEDEF	Discard type definitions (e.g. typedef's in C/C++).
-UNION	Discard C style union definitions

Languages

In the following description, a summary of each of the supported languages is given, together with the default file extension mappings and the tag entities which may be generated by crtags.

Language	**File extensions**
Ada	.a
Assembler	.asm, .s
Basic	.bas
C	.c
C++	.C, .c++, .cpp, .cxx, .h, .h++, .hxx
CRUNCH	.cr
Fortran	.f, .fcm, .f90
HTML	.html, .htm
IDL	.pro
Java	.java
Pascal/Borland Delphi	.pas

SQL	.sql
TeX	.tex
Perl	.pl, .pm
Verilog	.v, .verilog
VHD	.vhd, .vhdl
Yacc grammars	.y

Note that crtags uses a fuzzy-parsing`fuzzy parsing` mechanism to scan source files. This has the advantage of working in spite of any syntax errors and avoids the complexity of worrying about compile-time constant definitions which may affect the flow of the parsing (e.g. in the presence of #ifdef constructs).

The down-side of this is that the parsing may not be 100% correct as seen from the point of view of the compiler. The aim is to provide a level of accuracy which makes the tool useful to you.

The CRUNCH Language

The crunch language is the language used to write macros for CRiSP. The crunch language looks and feels a lot like the C language, and this should help users who are writing macros for the first time, but there are significant differences, which the user should be aware of.

The CRiSP language supports a number of primitive data types(pg. 15).:

32-bit integers (int)
64-bit floating point numbers (float/double)
strings (string)
lists or arrays (list)
structs

CRiSP acts as an interpreter for the language. The programs which the user writes are first compiled to a compact pseudo code format. Although CRiSP is designed to run as fast as possible and use as little CPU resources as possible, the design of the interpreted language is aimed at keeping the size of the macros as small as possible. Writing macros in the crunch language allows the internal architecture of CRiSP to be extended and improved upon in the future whilst maintaining compatibility, at the source code level for user written extensions to the editor.

The other advantages of writing macros in the crunch language is that the macros are totally machine independent, working equally well on Windows or Unix platforms. Macros are also convenient when you do not have access to a C compiler or other architecture specific development tools.

The crunch compiler is implemented using a full yacc grammar(pg. 25). of the ANSI C language, and although many constructs may be accepted by the compiler, they may not generate any code, wrong code, or cause the compiler to crash. When in doubt about the correct parsing of a macro, you should run the *crunch* command with the -c flag. This will compile the source code into the .m intermediate language file (the .m lisp-like language is CRiSP's assembly level code).

In order to write your own macros, you will need to understand various levels of detail. Writing simple one-off macros is easy, but there are a lot of details to learn if coding up complex multi-file macro packages.

1. The syntax of the language. The syntax is very close to ANSI C. For those of you who know this language, this means there is very little mental energy involved in understanding what to write or what to expect.
2. The semantics of the language. This covers the actual context dependent meanings of constructs within the language. This ranges from the meaning of a *switch* statement, to an understanding of the different data and variable types.
3. The internal data types and objects within the CRiSP language. CRiSP supports objects ranging from 32-bit integer values, to entire buffers containing edited files, callbacks, dialog boxes.
4. An understanding of the macro primitives CRiSP provides. CRiSP provides numerous functions which operate on internal data structures.

{button See Also, ALink(crunch,,,)}

The crunch compiler

The **crunch** program is the crunch compiler. It takes a source file (with **.cr** extension) and creates a **.cm** file, ready for loading into CRiSP. The crunch program uses its own internal preprocessor which is very ANSI-C like. By doing this gives the user more portability of macros and avoids common differences between standard preprocessors.

Next the intermediate file is converted directly to the binary output file. Crunch has a number of switches:

-c	Compiles the source file to a **.m** file. This is useful for understanding the translation process or to check for bugs in the compiler. If you have any problems understanding what crunch is doing, then use this switch.
-Dvar	Used to *#define* constants before preprocessing. This switch is passed directly to the preprocessor.
-f	Used to flush output during debugging. Causes the output to be written to the terminal. This is useful if crunch core-dumps and you want to try and ascertain at what point during code generation the problem is occurring.
-Ipath	Add a path to search for include files. This switch is passed directly to the preprocessor.
-g	Used to insert debugging information into the compiled code. This includes line number information, so that when a macro error occurs, CRiSP can report the line in error.
-m	This is the make flag. Tests the modification time of the output file versus the source file and only recompiles if it is necessary. This allows trivial makefiles to be built rather than having to face the bugs in standard make. (See the distribution makefile how to use this). For example, you can say: `crunch -m -o /macrodir *.cr` and only the out of date macro files will be recompiled. (No account is made of dependencies on include files).
-n	Print out the names of files which would be compiled, but don't compile them. This flag is useful with the '-m' (make) flag to verify what files will be recompiled.
-o file	Specifies the name of the output file to create. The file parameter can be the name of a directory in which case the output file is put into the specified directory.
-p cpp	Used to specify the path of the C preprocessor to execute if the one on your system does not conform to the standard used by the current Unix versions, e.g. if you are using Turbo C, or you have a POSIX compliant C compiler.
-q	If more than one source file is specified on the command line, crunch normally prints the name of each file as it is being compiled. This switch can be used to turn off this feature.
-S	Special non supported feature. Used to dump a symbol table.
-Uvar	Make the named variable undefined. Passed directly to the C pre-processor.
-V	Prints version number of compiler.
-#	Prints each pass of the compilation process as it proceeds.

The crunch compiler more or less understands the full ANSI C syntax, including structure definitions, bit fields and typedefs. However, *crunch* is really only designed to accept macros which can be used by CRiSP. At present CRiSP cannot handle structure and typedef definitions and so it is best to avoid these.

The crunch compiler normally creates a temporary intermediate file between the pre-processing stage and the compilation phase. Normally this file is created in /tmp. You can override this by specifying the name of a directory in either the CRTMP or TMP environment variable. (CRTMP will take precedence if both are specified).

{button See Also, ALink(crunch,,,)}

Data types

CRiSP supports a range of primitive data types and complex objects. The following is a summary of the basic data types:

Type	**Description**

int(pg. 21).	32-bit signed integer.
float(pg. 21).	64-bit floating point value.
string(pg. 21).	Variable length string.
list(pg. 22).	Arbitrary collection of objects, similar to an array. Lists may contain nested lists, and can be used like a structure or array.
declare(pg. 25).	Used to define a polymorphic variable - one that can contain a value of any type.

As well as these primitive data types, CRiSP also supports complex data types(pg. 38). used to refer to particular instances of objects within the editor.

{button See Also, ALink(crunch,,,)}

Variables - types, scoping, argument passing

CRiSP supports a minimal set of data types necessary to allow sophisticated editing macros to be written. Crunch requires that all variables to be used be declared before they are used. This is similar to the C language, and is a useful feature since it avoids bugs being introduced due to spelling errors. The compiler will complain about references to variables which have not been declared.

Although crunch may be classed as a fairly strongly typed language, it has mechanisms for processing arbitrary variable types. For example, a macro could be written to return the minimum value of the arguments passed:

```
int
min(int a, int b)
{
        return a < b ? a : b;
}
```

This is fine, but doesn't allow the user to write a generic macro which can handle arbitrary variable types. For example, if two strings were passed, maybe the shortest length string should be returned. This can be handled by crunch with variables which are called *polymorphic*. The term polymorphic means that a variable can have an arbitrary type and value. The type is dependent on its context. Originally, polymorphic variables were added to facilitate the processing of lists, which are sequences of values of arbitrary type. To write a more generic **min()** function, one could write:

```
declare
min(declare a, declare b)
{
        if (typeof(a) != typeof(b)) {
                error("Incompatible types.");
                return -1;
                }
        switch (typeof(a)) {
          case "integer":
          case "float":
              return a < b ? a : b;
          case "string":
              return strlen(a) < strlen(b) ? a : b;
          case "list":
              return length_of_list(a) < length_of_list(b) ?
                      a : b;
          default:
               error("Unknown type");
               return -1;
        }
}
```

Because the type of a polymorphic variable may change, CRiSP supports functions for determining the type of the variable and macros can ensure that they don't attempt to perform an invalid operation.

{button See Also, ALink(crunch,,,)}

Scoping

All variables created have a *scope* of visibility. CRiSP supports a number of scopes of visibility: static, local, global and buffer-local. Global variables are always available to macros and retain their values from one function to another. Local variables exist from the point at which they are defined to the end of the *current block*. The current block is defined as the current level of curly brackets. Static variables are variables which are local to a function but which maintain their value across calls to the function. (This is identical to the C mechanism). For example:

```
int fred = 99;
main()
{
        string  fred = "hello mum";
        int     i;


        for (i = 0; i < 99; i++) {
                list fred = quote_list(1, 2, 3);
                }
}
```

In this example, there are three occurrences of the variable **fred**. The first one is a global variable, and is assigned the value 99 when the macro is loaded. When the function **main()** is called, the global fred is saved, and a new variable is created, of type string. The *for* loop demonstrates a new occurrence of fred being defined purely for the scope of the loop. Within the loop, fred is a list. When the loop exits, the string version of fred is accessible. Eventually when the function terminates, the integer value for fred is accessible.

Internally, scoping is implemented by associating a block level with the definition of each variable. Global variables are defined in block 0, which is never exited. Conceptually, each time an open curly bracket is seen, a new level is entered. In the example above, the string version of fred is defined at block level 1. When the close curly bracket is seen the block level is decremented and variables defined in that scope are removed from the symbol table.

Because CRiSP is an interpreter, certain features of the language become available for very little interpretive overhead. One of these features is *dynamic scoping*. Dynamic scoping is similar to the scoping rules of **Pascal** rather than **C**. It is easiest to explain dynamic scoping together with an example:

```
int
func1()
{       int     a = 1,
                b = 2;

        func2();
}
void
func2()
{
        extern int a, b;

        message("a=%d b=%d", a, b);
}
```

In this example, the function **func2()** is called from **func1()**. The declaration:

```
extern int a, b;
```

is used to tell the crunch compiler that the variables **a** and **b** will be accessible at run-time, even if there is no definition of a and b within the current scope. Essentially, it just tells the compiler to not complain about undefined variable references.

When the line:

```
message("a=%d b=%d", a, b);
```

is executed, CRiSP searches the current block level for a definition of the variables a and b. Since these are not found, CRiSP then searches the current block level - 1. At this block level, the definitions are found.

When a variable is accessed, CRiSP needs to locate the symbol definition dynamically. The order of processing is as follows:

1. First a check is made for a static variable definition in the current function.
2. If no value is found as a buffer local variable, then CRiSP will try the current local variables of a function.
3. If no value is found as a static variable then CRiSP will try a buffer local variable.
4. If no value is found in the current stack frame then CRiSP will search all the nested stack frames, back to the outermost function call.
5. If no value is found then CRiSP will try for a global variable.

Note that it is possible to confuse CRiSP by declaring static variables inside local blocks (i.e. instead of at the top of a function definition). This confusion can arise when a nested block and an outer block define variables with the same name but with different attributes inside the nested block. Generally it is advisable not to redefine variables within nested blocks with the same name as an existing variable in an outer block to avoid any surprising results. (Note that this is only applicable to within a function; across function calls, symbols may have the same name, so you do not need to know how a calling function is implemented). These problems can arise because CRiSP is an interpretive language and doesn't necessarily assign unique addresses to variables as might happen with a compiled language.

{button See Also, ALink(crunch,,,)}

Argument Passing

CRiSP supports a special form of argument passing, known as lazy evaluation. The arguments to a function are not evaluated at the time a function is called (as it is in C). Instead they are evaluated at the time they are referenced in the called function. This can lead to some difficult to understand code and hard to find bugs, so it is important that the user understand this concept. This feature offers a lot of flexibility.

Before showing an example of this lazy evaluation scheme, it is necessary to discuss the mechanism used to implement argument passing. Writing a function which takes parameters, and calling that function looks and mostly feels like 'C'. For example, to define a function which takes three parameters, the first an integer, the second a string, and the third a list would look like this:

```
int
func(int arg1, string arg2, list arg3)
{
        ...
}
```

The code above is treated as if the function was implemented as follows:

```
int func()
{
        int     arg1;
        string  arg2;
        list    arg3;

        get_parm(0, arg1);
        get_parm(1, arg2);
        get_parm(2, arg3);

        ...
}
```

The user can pick either form for functions. Normally it is best to use the pure C style for function definitions, and use the **get_parm()** primitive when a non-C compatible calling sequence is required, or for *varargs* support.

For example, consider a macro which adds up all the integer parameters passed to it:

```
int sum()
{       int     arg_no = 0;
        int     sum = 0;
        int     arg;

        while (get_parm(arg_no, sum) > 0)
                sum += arg;
        return sum;
}
```

Crunch performs limited prototype validation and this is designed to catch inconsistent coding errors. You should therefore always specify prototypes for external functions so that CRUNCH can check that arguments agree. (CRUNCH cannot perform a complete type-safe check because variables can be declared as polymorphic, in which case the type of variable will not be known until run-time). In crunch, it is possible to indicate that an argument may be optional. This is done by preceeding the type specifier with a tilde:

```
int
func(int arg1, ~list, string arg3)
{
        ...
}
```

This causes the variables arg1 and arg3 to be set up on entry to the function, but it is the functions responsibility to get the value for the second parameter.

Given the above descriptions, it is now possible to understand the lazy evaluation scheme more easily. Consider the following program:

```
find_strlen(string str)
{       int     i = 0;
        int     len;

        len = iterate_strlen(str, ++i);

        /* At this point i is 1 greater than len. */
        message("len=%d i=%d", len, i);
}
int
iterate_strlen(string str, ~int)
{       int     len = 0;
        int     arg;

        while (1) {
                get_parm(1, arg);
                if (substr(str, arg, 1) == "")
                        break;
                len++;
                }
        return len;
}
```

In the call to the function **iterate_strlen**, the second parameter is specified as **++i**. This does not cause **i** to be incremented until it is **referenced** in the function **iterate_strlen()**. This occurs at the line:

```
get_parm(1, arg);
```

Lazy evaluation is used in the supplied macros mainly to allow specifying a private key mapping for pop up windows. For example, the function **select_buffer** takes an optional parameter, (the 4th one), which is not evaluated directly on entry to the function, but after the keyboard mappings have been set up. This allows the calling macro to specify the name of a function to call to do whatever is necessary just before the user is shown the popup window.

{button See Also, ALink(crunch,,,)}

Variable argument lists

CRiSP supports a variety of mechanisms which allow for variable numbers of arguments to be passed to macros. As described above the *get_parm()* and *put_parm()* primitives are used to access arguments to macros. A useful addition to these primitives is the **arg_list()** primitive. This primitive returns a list representing the arguments passed to the calling macro. This list can then normally be used as an argument to further macros to allow for the fact that the called macro may have been passed an arbitrary number of arguments.

In order to understand this clearly, let's take an example. Suppose we wish to write a macro which acts as a *wrapper* around an existing primitive, e.g. the **insert()** primitive. The insert() primitive is used to insert text strings into the current buffer. It can take an indefinite number of arguments, the first of which can optionally be a printf-like formatting string. We could use the **arg_list()** macro like this:

```
example()
{
    my_insert("hello %s", "world");
    my_insert("%d+%d=%d", 1, 1, 1+1);
}
/* Our function -- note no arguments are specified */
/* in the definition. */
my_insert()
{
    insert("[");
    insert(arg_list());
    insert("]");
}
```

In this example we access the variable number of arguments using *arg_list()* and thus pass on the arguments to the insert() primitive without needing to write any macro code to get at and pass on the arguments.

{button See Also, ALink(crunch,,,)}

Returning values and parameters

There are two ways to return values from a function: you can **return** a value as the result of the function, or you can modify one or more of the calling parameters (as in pass by reference).

Returning values

To return a value, use the **return** statement. Functions can be declared as **void**, indicating that no value is to be returned (i.e. the function is procedural). In which case, the **return** statement takes no argument. Falling off the end of the function is the same as executing a **return** statement with no value:

```
void print_message(string str)
{
      printf("%s\n", str);
}
int add2(int a, int b)
{
      return a + b;
}
```

CRiSP also supports an older archaic function, **returns**, which acts like a function call and arranges a value to be returned when the function exits. This function should be avoided where possible as it is not guaranteed to work if any other function or primitive is called after it. This is present for backwards compatibility only.

Pass by reference

Returning a value from a function using **return** probably accounts for 99% of the parameter passing mechanisms used in the CRiSP macros. The alternative way to return values is pass by reference. The syntax for this is:

```
void get_max(int a, int b, int c, int& d)
```

```
        {
                d = max(a, b, c);
        }
```
Note the ampersand after the type specifier for the last parameter in the list. Variables which are passed by reference are noted by the crunch compiler and any assignments to these variables cause the *right* thing to happen, i.e. the callers argument is updated.

The pass by reference mechanism shown above is actually implemented using the lower level **put_parm()** primitive. **Put_parm()** is a special macro primitive which lets you assign values to the calling functions parameters. The function takes two arguments - a number indicating which parameter to update and a value. The above example could be written as:`put_parm`

```
        void get_max(int a, int b, int c)
        {
                put_parm(3, max(a, b, c));
        }
```
The call-by-reference mechanism is new in CRiSP version 6, so many of the existing macros supplied with CRiSP still use this older mechanism.

If you call the above function without specifying an appropriately typed variable for the return value, then you are likely to get a macro error at run time or some other undefined behaviour. For example:

```
        get_max(1, 2, 3, 4);
```
will result in an error because the **4** being passed is not a legal value to which CRiSP can assign a value to.

{button See Also, ALink(crunch,,,)}

The int data type

The **int** keyword is used to declare integer variables, i.e. variables which can hold only integral values. CRiSP currently only supports 32-bit integers (i.e. chars and longs are not supported nor their signed/unsigned counterparts). Integer variables are 32-bit twos complement numbers. The 32-bit word size is chosen for maximum portability and usefulness.

Integer variables are used for many reasons -- as counters, indices, buffer identifiers, etc. The full complement of C operators are supported for manipulating integer variables.

Integers are always stored in macros in a machine independent fashion. This means that compiled macros using integer variables are portable to machines with different byte orderings.

{button See Also, ALink(crunch,,,)}

The float/double data type

The **float** and **double** data types are supported to facilitate implementation of macros which need to use floating point numbers, e.g. the calculator macros, and the **sum** macro. CRiSP has no internal use for floating point numbers.

CRiSP stores floating point numbers using the native C compilers *double* keyword, normally corresponding to a 64-bit quantity.

Floating point numbers are declared using the **float** or **double** keywords. Currently these two keywords are treated as being identical. It is recommended that users use the **float** or **double** keywords as appropriate to the task in hand. Later versions of CRiSP may support a shorter floating point type for efficiency.

Floating point constants compiled into macros are NOT stored in a machine independent manner, and thus compiled macros may not be portable to different machines.

Floating point numbers may be implicitly cast into integer values under certain circumstances.

{button See Also, ALink(crunch,,,)}

The string datatype

CRiSP supports a dynamic string data type. Strings variables may be used to store arbitrary length strings (up to 64K on 16 bit machines and 4GB on 32-bit machines). Strings may be used to store any sequence of characters, although storing the NULL character (ASCII 0) may cause problems, e.g. when determining the string length. `string:definition`

Storage for strings is dynamically allocated so no space needs to be preallocated for them.

Strings may be combined with the other data types to perform concatenation.

String constants are specified by enclosing the string within double quotes. For example

```
        "the help text"
```

You can include a double quote character by quoting it with a backslash as in:

```
        "Select the \"Help\" button for more help"
```

You can use the backslash character to quote the meaning of the next character, e.g. a newline. Specify two backslashes to get a single backslash. In addition, CRiSP supports the standard C style character abbreviations for specifying newlines, backspace, etc.

If you have a long string literal, you can make formatting of the code more pleasing by using implicit string concatenation. This is performed by specifying two string literals adjacent to each other. For example, the following two examples are equivalent:

```
        "The filename" "was not found."
        "The filename was not found"
```

Alternatively you can use the string concatenation operator (+)`string:concatenation` to achieve the same effect, but this is performed at run time rather than at compile time, and hence is slower.

{button See Also, ALink(crunch,,,)}

The list (array) datatype

CRiSP supports a number of data-types of which one of the most interesting and useful is the *list* data-type. A list is an extensible data structure. A list can be used to group other data items together so that a single variable can be used to refer to a whole collection of variables. In some instances in the CRiSP macro support code, lists are used as if they were arrays (the syntax for referring to lists can use the same notation). In other instances, lists can be treated like C structures.

A list is extensible, meaning it can grow as needed, e.g. by appending items to the end of it. A list can grow to any size less than 64K bytes in total.

A list may be used to store any other data type, including lists. A list may be extended only at its end, by appending data to it. Any element may be referenced by specifying its ordinal position in the list.

For example:

```
                    list    lst;

                    /* Assign initial value to a list. */
                    lst = quote_list(1, 2, 3);

                    /* Now add something to it. */
                    lst[3] = "hello";

                    /* Now modify element in the middle. */
                    lst[2] = quote_list("abc", "def", 1.2);
```

List elements (or *atoms*), are indexed using a zero offset, i.e. the first element in a list is accessed as **list[0]**.

Lists are *first-class* objects in the CRUNCH language. This means that many primitives can manipulate lists or be passed lists where appropriate. As illustrated above, lists are declared using the **`list`** data declaration. A list declared like this is akin to an array which is defined without an upper bound in C, but is automatically extended as needed. Lists can be manipulated in many ways.

Attempting to access negative indices of a list/array will cause an integer value zero to be returned. Attempting to access beyond the end of an array as an *rvalue* will return the value NULL. Attempting to access beyond the end of a list/array as an *lvalue* (i.e. perform assignment to an element) will cause the list to be padded with NULL values in the missing positions.

The following sections describe various aspects of list management in more detail:

{button See Also, ALink(crunch,,,)}

List assignment

Assignment may be used to *copy* one list to another, or to clear out a list (thus freeing its internally allocated storage). For example,

```
list    lst1;
list    lst2 = {1, 2, 3};


lst1 = lst2;
```

In this example, two lists are defined, the second of which is initialised at the point it is declared. The statements:

```
        list lst2 = {1, 2, 3};
and     list lst2 = quote_list(1, 2, 3);
```

are equivalent. The curly-brackets initializor get translated by the CRUNCH compiler into the functional form. The curly-bracket intializor format is more familiar and easier to grasp to C programmers, and is especially useful when defining lists which contain sublists. For example:

```
list    lst = { 1, 2, 3,
              {40, 41, 42},
              5};
```

defines a list with **five** elements in it, the fourth of which is a sub-list of three elements.

Lists can be built up into long data structures by appending data to them. The memory allocated to a list can be freed simply by assigning the value **`NULL`** to it:

```
list    lst = {1, 2, 3};


lst = NULL;
```

A null list is one whose length (as returned by the *length_of_list* primitive is zero). An uninitialised list is implicitly assigned the value NULL.

{button See Also, ALink(crunch,,,)}

Making Lists

A list can be extended by simply using the binary operator '+'. For example, you can concatenate a single item to the end of a list or add a new list at the end:

```
list    lst;

lst += 1;       /* lst == {1}           */
lst += "hello"; /* lst == {1, "hello"} */
lst += lst; /* lst == {1, "hello", 1, "hello"} */
```

As well as defining lists piece-meal as shown above, lists can be created using two primitives -- *quote_list()* and *make_list()*. *quote_list()* is a function which takes an arbitrary number of data types and returns a new list. None of the arguments are evaluated.

The *make_list* primitive is similar to *quote_list()* but each argument is evaluated in turn. For example:

```
list qlst. mlst;


qlst = quote_list(1, qlst);
mlst = make_list(1, qlst);
```

In this example, the list **`qlst`** will contain two elements -- the number '1', and the *string* 'qlst'. Remember that none of the arguments are evaluated. Care needs to be taken with this primitive to avoid confusion. If the value '1' had been replace by the expression, '1+2', then the resultant **`qlst`** would still have a length of two with the first element have a value of 3. If, however, the expression '1' above were to be replaced with a nonsensical expression such as '1+qlst', then the result qlst would still be a list of length 2, but the first element would be a sub-list representing the expression 'one plus qlst'.

Now consider the *make_list()* example. In this case, the assignment to **`mlst`** would be a list of length **three**, because each argument would be evaluated in turn. In this case the expression B`qlst` is a list of length two as defined in the previous statement.

You should try and understand these two very useful and powerful primitives because in many cases in the CRiSP macros, lists are passed around, either to represent static strings in a menu (in which case *quote_list()* is called) or as an easy means to pass variable length arguments to a function in an extensible manner (in which *make_list()* is normally used).

{button See Also, ALink(crunch,,,)}

Manipulating List Items

CRiSP provides a fairly rich set of primitives for manipulating list items. Individual elements in a list can be accessed either with the function **`nth()`** or by using square brackets, which are normally easier to read. (The CRUNCH compiler converts the square bracket notation to a call to the function *nth()*).

```
list    lst = {1, 2, 3};

message("lst[2]=%d", lst[2]);
```

Note that indices to elements in a list are zero based. The example above would print the message "lst[2]=3".

Individual elements or sequences of elements can be deleted using the primitive *delete_nth()*. This primitive takes two or three arguments, the first of which is the list to operate on, and the second is the index of the first item to delete. If the third argument is present then this can be used to indicate how many consecutive elements to delete. For example:

```
list    lst = {1, 2, 3, 4};


lst = delete_nth(lst, 2);
/* lst == {1, 2, 4} */
```

{button See Also, ALink(crunch,,,)}

Sorting Lists

A list of strings can be sorted into alphabetical order using the *sort_list()* primitive. The order of the sort can be controlled for increasing or decreasing alphabetical order.

{button See Also, ALink(crunch,,,)}

Searching Lists

Lists can be searched for strings and regular expression, using the *re_search()* primitive. Any non-string elements in the list will be ignored.

{button See Also, ALink(crunch,,,)}

Informational Lists

CRiSP contains various primitives which return lists as their return value. These primitives allow the macro programmers to enquire about the state of various objects within CRiSP.

Primitive	Description
bookmark_list	List of bookmarks (placeholders).
command_list	List of primitives built into CRiSP.
dict_list	List of all symbols defined in a dictionary.
file_glob	List of files matching a wild card pattern.
key_list	Get keyboard bindings.
list_of_bitmaps	List of all bitmaps and pixmaps in a .xpl file.
list_of_buffers	List of all buffer IDs.
list_of_dictionaries	List of all object dictionaries..
list_of_keystroke_macros	List of all defined keystroke macros.
list_of_objects	List of all user defined dialog boxes.
list_of_screens	List of all screens (peel off windows).
list_of_windows	List of all windows in the current screen.
macro_list	List of macros defined.

{button See Also, ALink(crunch,,,)}

The declare datatype

The **declare** keyword is used to create a polymorphic variable. A polymorphic variable is one in which the type of the variable stored can be changed. These are normally used as function parameters when it is not known until run-time what the actual type will be, or for looking at elements in a list. The actual type of a polymorphic variable is *frozen* when a new value is assigned to it, and until a new value is assigned the function can treat the type of the variable as if it were of the type frozen. For example:

```
declare         var;


var = 1.0;
var += 2.3;
/* var now contains value 3.3 */


var = "string";
var += "fred";


var = NULL;
/* Variable contains no value. */
```

{button See Also, ALink(crunch,,,)}

Structures

CRUNCH supports a minimal 'struct' facility. A structure is represented internally as a list but the usual X.Y syntax allows convenient access to elements of a list/structure without having to manually **#define** indices. Structures can be nested as in C. The order of definition of members of a structure is used to access particular indices into a list structure. There is no concept of 'structure' padding as a CRUNCH structure is not directly mapped on to a memory block.

{button See Also, Alink(crunch,,,)}

Language Grammar

The CRUNCH language is very similar to ANSI C. The following sections describe features of the language grammar:

→ Declarations(pg. 26).

→ Function definitions(pg. 27).

→ Expressions(pg. 27).

→ Loop constructs(pg. 28).

→ Conditional test - if(pg. 29).

→ Selection: switch(pg. 29).

{button See Also, Alink(crunch,,,)}

Declarations

There are two main types of declarations -- function definitions and data declarations. A function definition defines the body of a function. A data declaration is used to declare global variables or specify prototypes. A data declaration has the form:

[storage_class_specifier] [type_specifier] [declarator [= initializer]] ;

This is similar to C. The *storage_class_specifier* is used to identify how the variable is to be stored. The currently supported and meaningful storage class specifiers are:

extern	The variable is defined somewhere else. References to the variable will be validated against its type information, but no code will be generated to create the variable. This is typically used to implement a *forward* reference mechanism.
static	When applied to a macro definition (function definition), the macro can only be invoked from a macro defined within the same file. The macro function will not be visible or accessible to any other macros or for use in callbacks. The use of the **`static`** keyword is encouraged for all functions which are part of an implemented feature but of no use to other functions, and also for functions which are not going to be called back, e.g. as a result of a trigger, or keystroke.
	When applied to a variable defined within a function, static has the same meaning as in the C language, viz. the variable will maintain its value across function calls. A static variable can be initialised, in which case the function will be initialised the first time the function is called.

Type specifier should be one of the following:

int	Used to define a variable which will store a 32-bit integral value or to specify a function which returns an **int** value.
float	
double	Used to define a variable which will store a 64-bit floating point value or to specify a function which returns a **float** value.
string	Defines a variable which can store an indefinite length string or a function which returns a **string** value.
list	Defines a variable which can store a list value or a function which returns a **list** value.
declare	Defines a polymorphic variable which can store any data type, or a function which can return any type.
void	Used to indicate a function which doesn't return a value.

A *declarator* is defined as one or more variable names, or function prototypes separated by commas.

An initializor is used to give a variable an initial value, similar to C. Initializors may be arbitrary expressions, i.e. they are not limited to constant expressions, even for global variables. Lists may be initialised somewhat similarly to C structure initialisors.

For example:

```
extern list fred;
int     func (int, string, string);
int     a, b = 1;
```

```
list    l = {
        "Item-1", {1, 2, 3},
        "Item-2", "hello mum",
"Item-3", 3.14159, /* Trailing comma optional */
        };
```

{button See Also, ALink(crunch,,,)}

Function definitions

A function definition has the form:

```
[storage_class_specifier] [type_specifier] function_name
        ( [arg_list] )
        {
                function_body
        }
```

The *storage_class_specifier* is currently ignored, although a future version of the language will be able to understand the **static** class specifier. The *type_specifier* is used to indicate the return type of the function. This is currently ignored.

The argument list should be specified in the ANSI-C format, specifying the type specifiers and optional names. In addition, crunch supports the syntax:

```
~ type_specifier [name]
```

which acts as a place holder for an optional argument or an argument which is to be handled with different semantics from the standard C-style.

Crunch also supports the ellipsis (...) to indicate that optional further arguments may be specified.

Please note that crunch does not currently check function calls against prototypes.

{button See Also, ALink(crunch,,,)}

Loading a macro: main, _init

As explained below, global variables may be initialised with non-constant expressions, unlike C which is limited to a constant expression. Because of this facility, CRiSP provides a mechanism for ensuring that these global variables are initialised before the macros execute. All global variable definitions and initializors are compiled into a function called **_init**. When a compiled macro file is loaded (the .cm file), this macro is executed first. Programmers can put their own one-time initialisation code in the function **main()**. All the code in **main()** is executed within the context of the function **_init** after the global initialisations. To understand this better, it is best to compile your code with the **-c** switch and look at the lisp code that the compiler generates.

{button See Also, ALink(crunch,,,)}

Expressions

The following table summarises the operator precedence and associativity of the primitive elements of an expression. This table is a copy of the table which can be found by executing the **hier** macro at the command line prompt:

```
Arity       Operator                                    Assoc
---------------------------------------------------------------
binary   ()  []  ->  .                                  l -> r
unary    !   ~   ++  --  -  (type)  *  &  sizeof        r -> l
binary   *   /   %                                      l -> r
binary   +   -                                          l -> r
binary   <<  >>                                         l -> r
binary   <   <=  >   >=                                 l -> r
binary   ==  !=                                         l -> r
binary   &                                              l -> r
binary   ^                                              l -> r
binary    |                                             l -> r
binary    &&                                            l -> r
binary    ||                                            l -> r
ternary  ?:                                             r -> l
binary    = += -= *= /= %= >>= <<= &= ^= |=             r -> l
binary    ,                                             l -> r
--------------------------------------------------------------
                                                From K&R, p 49
```

In the above table, the following are **not** supported: **->**, **.** (dot), **(type)**, **sizeof**. Also, crunch does not support structures, unions, pointers, or explicit dereferencing.

Crunch supports a fair amount of automatic type co-ercion. The following table lists the coercion rules when used with the arithmetic operators only. (Type coercion is not performed for function arguments). The prefix character is used to identify the type of the variable or expression - i = integer, f = float, l = list, s = string, a = any type. **sym** is used to denote a symbol of the specified type; **expr** is used to denote an expression evaluating to the specified type. (Note that these conversions are implicit -- the typecast on the right hand side is not a supported feature, currently).

```
isym op= fexpr  => iexpr op= (int) fexpr
fsym op= iexpr  => fsym op= (double) iexpr
lsym += aexpr   => append aexpr to end of lsym


fsym++          => fsym += 1.0
fsym--          => fsym -= 1.0  etc..


iexpr op fexpr  => (double) iexpr op fexpr
fexpr op iexpr  => fexpr op (double) iexpr
iexpr + sexpr   => "iexpr + sexpr"
                   (string concatenation)
fexpr + sexpr   => "fexpr + sexpr"
                   (string concatenation)
aexpr + lexpr   => new list with aexpr at front
```

Basically, a string plus a number (or vice versa) converts the number to a string and performs string concatenation. A list plus any type creates a new list by concatenating list and value. An integer and floating point value when combined results in a floating point value. Using these rules, a value of one type can easily be converted to a value in another type:

```
string  s;
int     ival;

ival = 99;


s = "Value: " + val + " brass monkeys";
/* s = "Value: 99 brass monkeys" */


ival = 1000000;
s = "Value: " + (0.0 + ival) + " big macs";
/* s = "Value: 1e+06 big macs" */
```

When converting floating point numbers to strings, the "%g" printf format specifier is used.

{button See Also, ALink(crunch,,,)}

Loop constructs: for, while, do

There are three main looping constructs, the *for* loop, the *while*, and the *do* loop. The for loop is a generalised looping mechanism supporting the following syntax:

```
for ( init-expr ; while-expr ; post-expr )
     statement
```

The init-expr is evaluated first. Next the *while-expr* is evaluated, and if it evaluates to **TRUE**, then *statement* is executed. After *statement* is executed, *post-expr* is executed. The loop continues until *while-expr* evaluates to **FALSE**. Any combination of the *init-expr*, *while-expr*, and *post-expr* may be omitted, as in standard C. If *while-expr* is omitted, then the loop will execute indefinitely. In this case the only way to terminate the loop is if the *statement* part of the loop contains a *return* or *break* clause.

The *while* looping construct has the syntax:

```
while (expr)
    statement
```

expr is evaluated and if it evaluates to non-zero, then *statement* is evaluated. This process is repeated until *expr* evaluates to **FALSE**, or the statement clause causes an exit from the loop (either via *break* or *return*). While loops always evaluate the expr clause at least once.

The *do* looping construct has the syntax:

```
do
    statement
while (expr)
```

In this case, the *statement* clause is executed first, and the *expr* clause is tested after the body of the loop has been executed. Do loops are useful when you need to guarantee that the body of the loop is executed at least once.

{button See Also, ALink(crunch,,,)}

Testing expressions: if

The if statement is used to execute a piece of code conditionally. The syntax is:

```
if (expr)
    statement-1
[else
    statement-2]
```

The *expr* is evaluated and if it is non-zero, then *statement-1* is evaluated. If *expr* evaluates to zero (false), and if the *else* clause is present, then *statement-2* is executed instead.

Crunch also supports the tertiary '?..:' operator which can be used inside expressions, for example:

```
int     a = b > c ? b : c;
```

{button See Also, ALink(crunch,,,)}

Selection: switch

The switch statement is a compact and fast mechanism for selecting a statement to execute depending on the value of some expression. The general syntax is:

```
switch (expr) {
  case expr-1:
    statement-list-1
  case expr-2:
    statement-list-2
...
  default:
    statement-list-n
}
```

Switch statements look and almost feel like C switch statements. Switches are interpreted as follows: *expr* is evaluated and tested for equality against *expr-1*. If it matches, then *statement-list-1* is executed and execution continues after the switch statement. If the match fails, *expr-2* is tested, and so on until either a match is found, the *default* clause is reached or no entry is found. If no entry is found, then execution continues after the switch statement. If the default clause is reached, then the statements there are executed.

Crunch allows multiple cases to be associated with a single statement:

```
        switch (expr) {
```

```
                  case 1:
                  case 2:
                        do_something();
                        break;
                  case 3:
                  case 4:
                        do_something_else();
                        break;
                  }
```

The statement-list associated with a *case* statement may consist of zero or more statements, and may optionally be enclosed in braces, e.g. to allow declaration of local variables:

```
                switch (expr) {
                  case 1: {
                        int     i;
                        i = 3 * 4;
                        break;
                        }
                  ...
                  }
```

The *break* statement may be used to terminate the switch statement.

The switch statement has a bug associated with it. Consider the following example:

```
                int i = 0;
                switch (1) {
                  case 1:
                        i++;
                  case 2:
                        i++;
                  break;
                  }
```

In the C, language, *i* will have the value of 2 after executing the switch statement. In crunch, *i* will have the value 1. This is because crunch does not currently support the flow-thruoughfacility of C. This may be considered a bug, and user code should not rely on this as it is liable to change in future versions of the language and compiler. (The CRiSP supplied macros **do** rely on this feature and are wrong!).

{button See Also, ALink(crunch,,,)}

Debugging macros

CRiSP provides a number of facilities to aid in debugging of macros and tracing execution. There is no complete debugger environment but there are various mechanisms to help in tracing why a macro does not work properly. The following sections describe the features that are currently available.

Apart from the BRIEF compatible functions, these debugging features are subject to change in future versions of CRiSP. Eventually CRiSP will contain sophisticated macro debugging facilities, but the focus has been so far to improve the editing environment for end users.

→ Primitives for showing output.(pg. 30).

→ Macro tracing(pg. 31).

→ Debugging a function(pg. 32).

→ Debug on startup(pg. 32).

→ Debug buffering(pg. 33).

→ Monitoring variables(pg. 33).

Primitives for showing output

CRiSP CRiSP provides various primitives which are using when following the progress of a macro. These primitives and macros allow you to display messages to **stdout** or to the status area.

printf() This function is similar to the standard C library *printf*() function. It is used to display messages on <stdout>. This function is only really useful if you are using a GUI version of CRiSP, since otherwise the *printf*() output will destroy your screen and make it difficult to read the output.

The *printf*() function is useful for tracing a macro's execution, by sprinkling *printf*() functions within your macro.

message() and *error()*

These primitives are similar to *printf*() but write their output to the status line of the current window. This is fairly useful but because there is only one line for the status information one message will overwrite any previous message and hence this is not a good thing for fast output where you need to see a sequence of printed messages. The major difference between the *message()* and *error()* primitive is that one displays in the normal foreground color, and the other in the error color.

The other difference between *message()* and *error()* is the use of the *pause_on_error()* macro. This primitive can be used to pause execution of a macro whenever an error message is displayed, i.e. it affects the display of messages with the error() primitive but not the message() primitive. If pause_on_error() is called with no arguments then it toggles the pause state. When pausing is enabled messages displayed with the error() function will have a '..' appended to them. To continue execution simple press any character. This gives a primitive form of single-stepping.

status_message()

This is a macro, not a primitive, is a useful macro which operates like the *message()* primitive but displays a message on the status bar message area. To use this macro, simply call it but specify the first parameter as -1. The -1 indicates to use the status bar on the current dialog box. If no status bar is present then this macro performs a message() primitive for you.

Macro tracing

The *debug()* primitive is used to trace execution of a macro. When debug is enabled, CRiSP will log the execution of all macros to a debug file. By default this is located in the /tmp directory and is called /tmp/crisp.log. You can override the place where this file is created by specifying the environment variable $CRISP_LOG and setting it to the name of a file where debug is written to. (If the /tmp directory does not exist, e.g. for a Windows system, then the current directory is used by default).

There are a number of different ways to use the debug() primitive. You can either just execute the command at the Command: prompt - useful for debugging an interactive session, e.g. where you select menu options or hit toggle buttons etc. Alternatively you can embed the debug() primitive in your own macro to trace specific pieces of code.

If debug() is called with no parameters then it simply toggles the debug mode. When debug is turned on, any existing crisp.log file is deleted/truncated. The output of the debug command is a trace all macro primitives executed together with some extra information. The top of a debug file contains various environment information usually needed for technical support. The first part of the file is not really relevant to end-users but is useful when forwarding the file to a support person.

The trace output which is logged is the execution of the underlying '.m' language. Remember that crunch macros are compiled into an internal lisp-like format, and it is this which is the assembly language of the editor. For example, here is a very small sample of debug output:

```
CRiSP DEBUG ENABLED: sunos41 CR_SUNOS41 v4.2.0d
Environment:
CRCONFIG=/home/fox/.Crisp
CRFILE=newfile
CRHELP=/home/fox/crisp/help
CRPATH=/home/fox/crisp/x11/../macros;.;/home/fox;/usr/local/crisp/macro
s
CRROOT=/home/fox/crisp
CRTERM=xcrisp-col
HOME=/home/fox
PATH=/home/fox/bin.sun4:.:/usr/etc:/home/fox/bin:/usr/ccs/bin:/etc:/usr
/sbin:/usr/local/bin:/bin:/usr/bin:/usr/etc:/sbin:/usr/ccs/bin:/usr/ucb
```

```
:/home/fox/bin:/usr/openwin/bin:/usr/openwin/demo:/usr/local/crisp/bin.
sun4:/develop/sun4/bin:/develop/scripts:/develop/scripts:/develop/scrip
ts/init.d:/develop/scripts/lmfdbase:/develop/scripts/logclient:/develop
/scripts/logger:/develop/scripts/newsbase:/develop/scripts/ticker1
TERM=xterm
End of Environment
*** DEBUG ON (0x0001) ***
04:.........  iACC=0
03:........  iACC=0
             iACC=0
Returning to macro: exec_macro
*** TRIGGER=REG_KEYBOARD ***
01:......(status_update 9 )
Execute macro: status_update
02:.......(int obj_id )
          (get_parm 0 obj_id )
            obj_id := 9
            iACC=1
          (int curscr perc col line )
```

The first section "Environment" up until "End of Environment" is for technical support and includes useful information about the current version of CRiSP. You can normally just ignore this section.

When debug is turned on the log file is annotated with the "DEBUG ON" message. Thereafter you can see each primitive as it is executed. The lines that say something like "iACC=" show the result of executing a previous macro or primitive. For instance, "iACC" means the accumulator contains an integer value. The internal execution engine has an accumulator which can contain integers, floats, strings, lists or NULL values.

Assignments to variables are shown by a line like:

```
obj_id := 9
```

A lookup of a symbols value is shown as an '=' as opposed to a ':='. The actual details and amount of information available in the crisp.log file is subject to change so you may encounter differences as future versions of CRiSP are released.

Debug applied to a function

One of the problems with the crisp.log file generated with the crisp.log file is the volume of information presented to you. To reduce the size of the output you can simply place debug() function calls in your macros closer to the point of the area of interest.

Another alternatively is to trigger the enabling of the debug information when a specific macro is executed. To do this, execute the debug command with the name of a macro as its parameter. For example:

```
Command: debug fred
```

This would turn on debugging as soon as the macro "fred" is executed.

Debug on startup

Sometimes you may need to turn on debug when CRiSP starts, e.g. because you may have written your own macro which gets loaded at startup and cannot get to the Command: prompt early enough to execute the debug command.

CRiSP provides a '**-d'** switch on the command line to turn on debug before the first macro is ever executed. Be careful: the **-d** switch is an acceptable abbreviation for '-display' and if you are using an X11 version of switch the X11 libraries will intercept the '-d' switch if it is followed by *anything*. This usually results in a cryptic error message of:

```
Error: Can't open display: ....
```

CRiSP supports a **-debug** switch as an alias for **-d** to overcome this problem. Alternatively, ensure the -d switch is at the end of the command

Debug buffering

Output written to the crisp.log file is normally performed using buffered I/O. Sometimes it may be necessary to cause CRiSP to flush the output buffer as soon as any data is written to the file. This is most normally needed when CRiSP is core dumping and the last buffer has not made it to the disk file.

You can cause output flushing to occur by specifying a -1 parameter to the debug primitive. When the -d switch is used on the command line, you can enable flushing by using the '-f' switch. E.g: "**crisp -df**" will start the Motif version of CRiSP and enable debugging, with buffer flushing enabled.

NOTE: Enabling flushing will significantly slow down the execution of CRiSP.

The vars() macro

The vars() macro is a utility function which you can execute at the Command: prompt to display the current values of all *global* variables. This can be useful if your macro is using global variables, but is not so useful if you want to see local variables. The vars() macro is designed to be run after a macro has been finished, which is why it can only display the state of global variables.

Note: the vars() macro will not show the value of *static* variables.

Buffers, Files and Windows

In order to understand CRiSP better, it is important to understand the basic concepts of buffers, files and windows as they are used within CRiSP.

A buffer is a way of manipulating files. A buffer is created when a file is edited. It stores the entire contents of a file, and keeps track of the changes being done to the file. The user can make arbitrary changes to the buffer without actually destroying or modifying the original file. Buffers have an undo-list associated with them. The undo-list allows the user to undo any editing operation applied to the buffer. CRiSP can keep track of an unlimited number buffers at any time. Each buffer has a set of attributes(pg. 34). which is used to keep track of the status of the buffer. These attributes may be manipulated by the macro language.

A file has the obvious meaning associated with the underlying filing system. Files can be read into buffers and manipulated, and not until the buffer is written away is the file on disk actually modified. This allows the user to maintain a long editing session and only when the user is satisfied with the changes is the file updated. Also, if the system crashes during an editing session, the original file will not be damaged. CRiSP contains facilities for keeping backup copies of files, so that even after an editing session the user can go back to previous versions of the file.

A window is a way of seeing a part of a buffer on the screen. On startup, only one window is visible, occupying the whole of the screen. This initial window can be split (known as tiling because the windows always cover the whole of the screen and abut each other). Windows can be split and new windows created. The only limitation is the size of the screen. If you have too many windows on display, they will be too small to display any meaningful part of a buffer.

Each window is independent of the others, and the user can select a window and pan around a buffer. Different windows can display different parts of the same buffer; for example, the user may be looking at the declarations at the top of a C program, and modifying a piece of code in another window.

CRiSP also supports popup windows. Popup windows are typically used to display information on a temporary basis, e.g. a list of buffers currently being edited, or a help menu. Popup windows obscure the background windows normally used for editing. Popup windows are normally created by special purpose macros. CRiSP contains many macros which use popups to display data.

CRiSP stores all buffers and files in memory; therefore, CRiSP is limited to editing files which are no larger tan the amount of swap space free. In practise this limitation is very rarely a problem. Most virtual memory systems support more swap than most of the standard editors can actually use. For example, vi is limited by the number of lines in the file rather than amount of memory available, and Emacs is limited to a maximum of 16MB for all editing. CRiSP can cope with very large files on most systems.

There is no limit to the number of files, buffers or windows which can be created at any one time (except for swap space). Also, files have **no** line length limitation. This makes CRiSP useful for editing certain types of files which many other editors or standard utilities cannot cope with.

Buffer Attributes

Each buffer has a number of attributes or modes associated with it. The following is a summary of these attributes, and the following sections explains the purpose of these attributes in further detail.

→ Ansi mode(pg. 34).
→ Backup flag(pg. 34).
→ Binary flag(pg. 34).
→ Buffer contents(pg. 35).
→ Buffer name, buffer ID and Filename(pg. 35).
→ Carriage-return flag(pg. 35).
→ Character maps(pg. 37).
→ Current cursor position(pg. 35).
→ Modified flag(pg. 35).
→ Permissions & Read-only Flag(pg. 35).
→ Process Buffers(pg. 36).
→ Region marker(pg. 36).
→ Symbol table(pg. 36).
→ System Buffers(pg. 36).
→ Tab settings(pg. 36).
→ Undo information(pg. 36).

Ansi Mode buffer attribute

Ansi mode is a special mode normally used with process buffers, in which Ansi escape sequences embedded within a buffer are interpreted and displayed correctly, rather than being displayed literally. For example, the string "<ESC>[34m" can be used on an Ansi conforming display (e.g. PC console, or VT-300 compatible terminal) to set the foreground color to dark blue. When ansi mode is set, all characters to the right of the escape sequence will actually appear dark blue. This facility is used mainly for process buffers where some program is run which uses Ansi color sequences and cursor movement facilities. This mode allows CRiSP to display the screen output in a window correctly. For example, it is possible to use the 'vi' editor inside a process buffer.

Ansi mode can be used independently of a process buffer, for example to review output from a program which uses these escape sequences.

Another possibility, not yet presently implemented in CRiSP, is to write a macro which changes the colors of keywords in the buffer, e.g. in a C program comments could be in green, and keywords in cyan. (This is really only feasible if the text within a buffer is easily parsable).

Another use is to look at the system manual pages inside a CRiSP buffer. Normally looking at the output of the *man(1)* command is a bit difficult because of the underlinings.

The **ansi** macro can be used by the user to turn on or off the ANSI attribute for a buffer.

The Backup Flag

When a file is read in to a new buffer, a flag is set, called the *backup flag*, which is used to indicate that when the buffer is saved a backup of the file should be made (either in the backup directory as specified by the set_backup() primitive or with a .bak file extension). This backup is only made **once** during each editing session. This allows the user to make lots of changes to a file and save it frequently and still be able to access the previous version of the file before editing started.

This flag may be set or cleared by a user macro, e.g. if the user doesn't want a backup made for a particular file, or maybe the user wants backups to be made after the file is written away each time.

This flag is used by the rebackup mechanism to allow changed buffers to force creation of new backups after a specified period. (Refer to the description of the *autosave* variable in the .crinit file in the User Guide.

The Binary Flag

When a file is read into a buffer, CRiSP tries to determine whether the file is a text file or a binary file. CRiSP can handle either, but response can become sluggish when lines are very long. (CRiSP can handle infinitely long lines, but it is expensive to compute how long they are for screen purposes). To make it easier to edit these files, CRiSP breaks up binary files into lines containing only 32 characters. To avoid problems if the file is modified and written away, CRiSP sets the binary flag. This flag means that the newline character normally written at the end of each line is not saved for binary files, and therefore the edited binary is what the user expects. If this flag were not set or if the user macro turns it off, then newline characters would appear at 32 character intervals in the output file.

Buffer Contents

A buffer may be considered as an array of lines, where a line is stored internally using a length + data. This means that each line can contain an unlimited amount of data and that any characters may appear within a line, even newlines (for example with binary files). CRiSP does not suffer the problem that *vi(1)* does of stripping out unprintable control characters.

CRiSP has no limit on the number of lines in a buffer (subject only to memory and disk space limitations). You can tweak CRiSP's memory utilization for large or pathologically shaped files using the **Options→Memory configuration** menu.

Buffer name, buffer ID and File name

Each buffer is identified by an integer number internally. Macros which manipulate buffers or switch between buffers use the buffer ID. Buffer IDs are not normally visible to users. Instead, they see a buffer name and a file name.

The filename is the name of the file stored in a buffer. Buffers don't necessarily have to have a file stored in them. Some buffers are used internally to store information about the current editing session, e.g. the scrap buffer. The filename is used by users to distinguish one buffer from another.

The buffer name is a sort of shorthand name for a buffer. Normally when a file is read into a buffer, the actual filename of the file is used as the buffer name. Different buffers may contain the same buffer name. For example, if the user edits the file /usr/fox/test.doc and /usr/fox/backup/test.doc, there will be two buffers called test.doc. The full-filename associated with each buffer allows the user to distinguish them. This full-filename is the file written to when the buffer is saved. The buffer name is the string printed at the top of the window when it is *attached*.

The Carriage-Return flag

This flag indicates whether the carriage-returns (CR) characters should be appended to the end of each line on output. This would typically be used for editing DOS text files which use the CR-LF sequence to terminate files, rather than the Unix standard of just plain LF.

When a file is read in, if CR characters are found before the LF at the end of the line, then this attribute will be set automatically.

Current cursor position

CRiSP maintains two separate cursor for each buffer - the normal cursor, as displayed on the screen, and a parallel cursor for process buffers. (See below for a description of process buffers).

If a buffer is not being displayed in a window, then the cursor position changes as a result of text insertions, deletions, pattern searches, etc. When the buffer is attached to a window, the cursor is copied into the windows data structure. This allows each window to maintain a separate view of a buffer.

When a buffer is detached from a window, the cursor is copied into the buffers data structure.

Modified Flag

Each buffer has a flag called the *modified flag*. This flag is set whenever a change is made to the buffer, e.g. an insertion or deletion. The flag is used to indicate that the buffer needs to be saved before terminating CRiSP, and is used, for example, by the autosave macro.

Permissions & Read-only Flag

When a file is read into a buffer, a note is made of the original protection bits on that file. This is needed so that when the file is written away, it can give the output file the same protection bits as the original file. For example, when editing an executable shell script, the execute bits are turned on for the modified file so that it is still executable.

If a read-only file is edited, then CRiSP does not allow any changes to be made to the buffer. This stops the user making changes and then finding out at the end of the editing session that they cannot be saved (a frequently annoying problem with *vi(1)*). The user can clear the read-only bit for a buffer by calling the **make_writable** macro (available from the features menu) or by typing <Ctrl-A><Ctrl-W>.

Process Buffers

Each buffer can have an external system process associated with it. These output from these process is automatically inserted into the buffer as it appears. Input from the keyboard can be directed to these processes. These process buffers are normally use to access the command line interpreter, e.g. the shell, so that instead of having to keep stopping CRiSP to access some system program, e.g. *make(1)*, the user can view output in a window. This is similar to having xterm windows under the X11 windowing system.

Process buffers are useful because all output from the process is easily accessible by scrolling the window using the normal editing keys, so that error messages, etc. which have disappeared from view can be seen.

Process buffers are supported by a number of macros to aid in using them.

These processes are sometimes referred to as pty's, because on systems that support them, pty's are used to implement this functionality. This term (as far as I know) originates from the ancient TOPS-10 operating system (circa late 60's). A PTY is like a normal terminal, except there is no physical manifestation of the terminal. Instead of input coming from a real user, the input comes from another program; output from the PTY can be read by the controlling program, e.g. CRiSP.

The pty facility in many Unix implementations is used where possible. Under other systems, pipes are used. Pipes are not as good as pty's, because for example it is not possible to run vi in a process buffer on systems where this is true.

Region Markers

Each buffer can have an optional *marker* or *anchor* associated with it. A marker is used to highlight a section of the buffer when it is displayed on the screen. This allows the user to highlight a section of text and then apply some operation to it, e.g. cutting it to the scrap. A buffer can only have a single marker at any one time, but each window which is displaying a buffer can maintain its own marker.

Symbol Table

Each buffer can have a set of user defined variables associated with it. These are referred to as *buffer local* variables. These variables exist only whilst the buffer is the current buffer. This facility allows macros to maintain their own data structures on a per buffer basis.

Refer to the description of the *make_local_variable* macro.

System Buffers

System buffers are special buffers which tend to be associated with the implementation of certain macros. System buffers tend to have two attributes which make them useful -- they do not store undo information, and thus buffer modifications are faster, and also they do not appear in the buffer list (available via <Alt-B>).

Because system buffers are marked differently to normal buffers, user macros can avoid accessing these buffers unless they themselves created them. For example, the autosave macro does not attempt to autosave the command history buffer.

In CRiSP, the system buffer flag and the undo-flag for a buffer are two separate buffers, allowing non-system buffers to have the undo turned off for a buffer.

Tab Settings

Each buffer has a set of tab stops defined for it. These tab stops are used when the buffer is displayed in a window. Having the tab stop as an attribute of a buffer allows different buffers and files to have different tab stops.

Undo Information

Each buffer has an undo list associated with it. The undo list allows changes made to the buffer to be undone. This feature can be turned off for any buffers. Turning off the undo feature can be useful for two reasons. Firstly, operations are faster if this information does not need to be saved. Secondly, disk space can be saved if a voluminous edit is to be made on a very large file, e.g. global translate.

The undo information is stored in a temporary file, which is not visible via the normal *ls(1)* command.

All undo information is undoable itself. This is termed *redo*.

Character Maps

A character map is a way of viewing data through a window. Character maps are used to implement the 'view' and 'literal' macros.

A character map is simply an array of strings corresponding to each of the possible 256 byte values. When a particular byte is to be displayed in a window, the corresponding string is displayed. For example, when the hexadecimal character 0x41 is to be displayed, the letter 'A' appears on the window. This is configurable, and if the user wants to use a non-ASCII character set, then the character map can be modified so that the EBCDIC entry for 'A' is used instead.

The 8-bit character set has four important regions:

Characters 0x00..0x1f (the control characters)

Characters 0x20..0x7e (the printable ASCII character set).

Character 0x7f (ASCII DEL)

Characters 0x80..0xff

All terminals support the ability to display the ASCII printable characters but few terminals provide facilities to display the other characters. In *vi(1)*, for instance, control characters are displayed as a caret followed by the non-control character. These characters take up two character positions on the screen.

CRiSP supports terminals which can print all 8-bit characters, e.g. PC terminals amongst others. Each user may have a preference for how these non-printing characters are to be printed, e.g. whether in C-style octal notation, hexadecimal notation, or maybe in the DEC editor EDT style (using keywords inside angle brackets, e.g. <TAB>).

Even if a user is happy with some means of displaying characters, he/she may find this scheme inflexible sometimes. For example, if the terminal does not support a full 8-bit character set, then control-A appears as '^A'. When viewing a binary file, it may be better to have each character occupying the same width on the screen so that it is easier to look at character offsets.

All this functionality is supported by the character maps. (For real examples of using the character maps, refer to the view.cr macro).

CRISP supports an arbitrary number of character maps. Initially CRiSP starts off with a default character map, where the non-printing ASCII characters are dependent on whether the terminal supports 8-bit characters or not. If not, control characters print in up-arrow format, and the characters with the top bit set print in the "\x" style notation.

Character maps are created via the **create_char_map()** primitive. This primitive is supplied with a list of strings, normally corresponding to each ASCII character position, i.e. a list of 256 strings should be specified to remap the entire display character set. Any characters not defined are inherited from the base character map. Character maps are given identifiers which are used with the other character map primitives.

Each character may have associated with it a flag. These flags are used to handle the TAB and BACKSPACE characters. It is not sufficient to map TABs to some fixed output string -- the tab character may need to be translated to a variable number of spaces. Likewise, in ANSI mode, the BACKSPACE character may need interpreting to implement bolding and underlining, e.g. when viewing the output of the 'man' command. The character map system is so generic that any character can be defined to be a TAB on display (even more than one at the same time).

Character maps provide a view onto a buffer, via a window. It is important to understand what character maps are doing because you can get into strange corners if they are not treated with respect.

Consider the case where you want to look at a binary file, e.g. a Unix directory. Unix directories are like normal files but include text (filenames) and binary data (*inode* numbers). Looking at a directory in pure text mode is untidy because of the variable width nature of the binary character set. Looking at a directory in pure binary mode makes viewing the filenames painful. What you can do is create two windows onto the same buffer -- one displaying in binary (type 'view hex'), and another viewing the file in ASCII (normal mode).

Each window and buffer has a character map associated with it. By default buffers have no character map associated with it. When they are displayed in a window the character map associated with the window is used to view the buffer. (Windows always have a character map associated with them). If a buffer has a character map associated with it (via set_buffer_cmap()), then this is used to override the window definition.

Associating character maps with buffers is important to understand because it controls the mapping between the externally visible column position in a line and the internal character pointer to the text. This is important because of the effect of inserting text in one window, and doing an undo in another which is using an alternate character map.

Having a fixed width character map can make it easier to write a hex mode editor to move from one character position to another.

Objects supported by CRiSP

CRiSP supports a variety of high level data structures, or objects, which can be manipulated within the macro language. Most of these you will already be familiar with by virtue of using CRiSP.

One of the most fundamental types of object within CRiSP is the buffer(pg. 33). Buffers are intimately tied up with files and windows. A buffer is a temporary holding area for a file whilst it is being edited, and a window is simply a view onto a file.

The following summary summarizes the different types of objects available within CRiSP.

Name	**Description**
buffer(pg. 33).	CRiSP is a file editor. Each file is stored in a temporary area called a buffer. The buffer object has numerous attributes, such as read-only status, undoability, current line number. Buffers are created from files, and after modifications, can be saved back to a file on disk.
bookmarks	A bookmark is a saved buffer position which allows you to save a position and jump back to it using a memorable name.
colorizers	A colorizer is a description of the syntactic elements of a language which is used so that when files of a particular type are edited, the appropriate colorization process will take place.
Colors(pg. 46).	CRiSP supports various operations on colors in order to allow various aspects of the screen display to be customised by the user. In addition color mappings are defined to allow the colorization of buffers to be affected.
dialog boxes	Dialog boxes are representations of the GUI dialog boxes used in many parts of the CRiSP user interface. Dialog boxes are very powerful and sophisticated objects and are described below in further detail. Dialog boxes consist of sub-objects, such as push buttons and input fields, which can respond to user actions.
Keyboard(pg. 87).	A keyboard is a mapping of function keys to macro functions and provide the basis for all editing activity. Keyboards can be made context sensitive so that functions are only invoked within the context of a particular buffer, for example.
Regions(pg. 40).	A region is a highlighted block of text associated with a buffer, usually used for cutting and pasting type operations.
registered macro(pg. 42).	Registered macros are macros which are invoked when particular events are triggered, for example, after an idle timeout, or when a buffer is modified or deleted. These macros are invoked automatically when the associated event takes place.
scraps	A scrap is a temporary buffer used for holding the contents of the cut and paste buffer. CRiSP supports multiple scraps.
screens	A screen is a GUI object which corresponds to a collection of windows with a dialog box. For example, when CRiSP comes up you are presented with a single top level window containing an editing area, as well as a status bar, scrollbar and menu bar. The editing area is an instance of a screen. If multiple top level windows are created (e.g. from the Windows->New Window) menu option, then each of these is a screen.

timer(pg. 45).	CriSP supports callbacks based on an elapsed real-time clock.
window(pg. 33).	A window is a character mode window with a view on to a buffer. A single screen can contain multiple character mode windows.

→Buffers(pg. 33).

→Macros(pg. 41).

File Types -- Text files and Binary Files

CRiSP allows any file type to be edited, whether the file is binary or text. Editing binary files is allowed although CRiSP does not contain any special facilities (yet) to perform editing on these files.

Binary files are identified by looking for non-printable characters in the first 100 or so characters of the file. If there are too many, CRiSP reads the file in, but makes each *line* only 32 characters wide. This avoids performance problems with CRiSP whereby if it did not do this, then it would read the whole file into a single line, making editing and scanning difficult.

Editing binary files can be useful, e.g. as an alternative to the *strings(1)* utility, to look at the printable strings in a binary. It can also be useful to make small edits to binary files. For example, to patch string literals in the file. Care is needed when editing executable files, since the file size needs to be exactly the same before editing the file as afterwards.

CRiSP notes buffers containing binary files, to ensure that an arbitrary newline character is not inserted at the end of each 32-character line.

Backing up Files

Backups are created when a file is actually saved. Every time a new file is edited, the old file is renamed so that even after writing a file away to the disk, the previous version of the file can be recovered. CRiSP can be configured to create backup files in a separate directory (useful for doing mass deletions when disk space is low) or for creating a '.bak' file in the current directory. To some users it may be distracting to have lots of directories filled with .bak files.

In addition, CRiSP supports the ability to keep multiple copies of old backed up files. Again, this is implemented via the set_backup() primitive. When this variable is set, crisp creates a set of sub-directories in the main backup directory, named .../0, .../1, etc (the higher the number the older the backup). When a file is written away, the most recent backup is put in the backup directory. The previous version in the backup directory is moved to the .../0 sub-directory. That is, the backup directory contains the most recent backup, .../0 contains the second most recent backup, and .../N-1 contains the oldest backup.

It is sometimes useful to turn off backup file generation for certain files, e.g. very large log files; you can tell CRiSP to turn off creating a backup file for the current file by using the *set_backup* macro.

In addition to the above two features, CRiSP has a builtin facility to save buffers in an emergency. If CRiSP detects an internal error (e.g. segmentation violation) all the currently modified buffers are written away to the current directory, with filenames called BUFFER-0, BUFFER-1, and so on. This should never happen, but bugs have a habit of laying dormant until the most unfriendly time.

To optimise performance, CRiSP uses the following algorithm when trying to backup a file:

1. If the file has multiple links to it, the file is copied to the destination directory. (See below). If the file has only a single link, then it may be link()ed to the destination directory.

   (Linking is much faster than copying because the actual contents of the file need not be copied -- just the file name is changed).
2. For each directory in the backup directory list try to link() the file to the specified directory. This operation may fail if the directory is on a different file system. (This step is skipped if the BACKUP_SLOW flag is set).
3. If step 2 fails, then for each directory in the backup directory list, try and copy the file to the resultant directory. This may fail because of permissions.
4. If steps 2 & 3 fail, try and create a file with a .bak extension. If the file has multiple links, then the old file is copied to the new file. If the file has a single link, then it is link()ed to the .bak file.

Autosaving

Autosaving is a feature enabled by default, whereby any modified files (buffers) within the current editing session are saved at 60 second intervals when there is no keyboard activity. This ensures that if you walk away from an editing session and the system crashes, that a copy of your work will be saved. The autosaved file is given a different name to the real file name to avoid prematurely committing changes to the file. These autosaved files are automatically deleted when the user exits from CRiSP.

CRiSP itself does not directly implement autosaving; instead, it is a macro which is supplied with CRiSP. CRiSP supports the mechanism for implementing the autosave feature. If any user does not like the mechanism, then he/she can modify the macro to make it do what is really wanted.

Autosaving is implemented via the idle timer. The idle timer is a timer which is maintained by CRiSP. The idle timer goes off when the user hasn't typed anything for 60 seconds. (This is configurable via the command line). The autosave macros looks at each non-system buffer, and writes out the buffer if it has been modified.

Autosaved files are written to a file with a different filename from the original to avoid destroying the original before the user is committed to keeping it. The actual filename used is operating system independent, and is a function of the real file name. Under Unix, if the file is called foo.c, then the file is saved in a file called **@foo.c@**. The use of the @'s is to make the files stand out when performing a directory listing. The prefix '@' is useful since if for some reason CRiSP does not delete the autosaved files, then it is easier and safer to issue an 'rm @*' command than an individual rm for each file. The trailing '@' is added on so that if the user issues a command such as:

ls *.c

then the autosaved files don't get included in the list. The major objective here is to avoid sticking on a long suffix (e.g. .asv) to a file name on systems such as V.3 Unix, which only have 14 character file names anyway). Adding suffixes to a filename is a dangerous game since the original filename may already be at the 14-character filename limit, and the suffix would get ignored. Having a prefix and suffix seems to be the best of both worlds.

Under DOS, Windows/NT or OS/2 with the FAT file system, a **.asv** extension is used instead, due to limitations on the lengths and styles of filenames.

These autosaved files are deleted when CRiSP exits.

When the autosave macro runs, it prints the message **Autosaving...** followed by **Autosave complete** when it has finished.

Core dumping

In the case the previous two mechanisms are insufficient to save the files you are editing on, CRiSP has a further *dramatic* file saving feature. If CRiSP detects an internal segmentation violation or other error which might cause CRiSP to die, then a special emergency macro is executed. This macro is implemented via the REG_INTERNAL registered macro. The code for this may be found in the core.cr file. It basically saves away all modified files (after prompting the user), to files called BUFFER.1, BUFFER.2, etc.

This macro is designed to be as minimal as possible and not rely on all aspects of CRiSP being able to work.

If CRiSP crashes it is most likely to be due to some strange combination of events which caused it to corrupt memory, and so the core macro has a good probability of succeeding in its attempt to save files.

Regions and markers

A region, or marker, is an area of the current buffer which can be manipulated independently of the rest of the text in the buffer. Most commonly regions are used to highlight a block of text which is to subsequently be deleted or copied to some other buffer. A region is created by typing one of the four keys: <Alt-A>, <Alt-C>, <Alt-L> or <Alt-M>. Each of these different keys creates a different type of region. When the cursor is moved, the area of text from where the marker was dropped and the current cursor position is called a *region*. Certain editing commands change their functionality whilst a region is highlighted, e.g. the <Del> key deletes the currently highlighted region, rather than the character the cursor is on.

Column regions are not implemented directly by CRiSP, but instead are manipulated by various macros. CRiSP only supports the screen drawing necessary to draw a rectangular marker.

There are four different types of regions, each one accessible from a different function key:

Key	**Description**
<Alt-L>	Drops a line marker. When a cut or copy command is issued, whole lines will be affected in the operation.
<Alt-C>	Drops a column marker. Text falling within a rectangular region from where the anchor was dropped to the current cursor will be highlighted.
<Alt-M>	Drops an inclusive marker. When a cut or copy command is issued, the character under the cursor will be included in the operation.
<Alt-A>	Drops a non-inclusive marker. When a cut or copy command is issued, the character under the cursor will not be included in the operation.

Macros

One of the most important features about CRiSP is that it is extensible, i.e. if you do not like some aspect of the user interface, you can change it without having to recompile the source code (usually). This extensibility is provided by macros.

A macro is a sequence of instructions which performs a high-level function. Different people have different ideas about what a programming editor should look like. Some people want simple options, others need complicated macros which can perform tasks as complicated as sort the functions in a .c file into alphabetical order.

CRiSP is designed to allow it to be easy to customise the editing tasks that a user wants, no matter how complicated the editing facility required.

Rather than complicate CRiSP internally with hard to change ideas about what facilities should be available, CRiSP provides a set of builtin primitives which manipulate the objects CRiSP knows about, e.g. files, buffer, windows, etc.

CRiSP has a programming language which allows users to combine these primitives into more complex functions.

For example, CRiSP provides a set of functions to search the current buffer for a string. One of the high level macros provided with CRiSP matches braces, i.e. it checks that there are an equal number of opening and closing brackets. This is an example of a macro. The macro is built from the primitives which CRiSP has compiled into its code.

If you do not like the way these macros are written, then you may easily customise them.

CRiSP provides a complete language for writing macros in. This language, called **crunch**, is similar to ANSI C. Experienced C programmers should have little problem understanding the macros supplied with CRiSP, and can use a lot of ideas from C in writing macros.

Global and Static macros

CRiSP supports the definition of global and static macros, in a manner similar to C. Using the CRUNCH compiler it is possible to define a static function simply by use of the **`static`** storage-class specifier keyword.

Because CRUNCH is an interpretive language it is necessary to be careful how you use static functions otherwise you may not achieve what you expect.

CRiSP maintains essentially two sorts of macro tables internally -- a list of global macros, and a list of static functions. There is actually a list of static function per macro file loaded.

When a macro attempts to execute a macro, CRiSP will check to see whether a macro of the desired name was declared as *static* in the same file that the calling macro was defined. If so, then the static macro will be called. If not, then CRiSP will try the global macro table. This is the ONLY way to invoke a statically declared macro, i.e. by a macro within the same .cr file attempting to call it. This means a user cannot directly or indirectly attempt to call a static macro, e.g. from the Command: prompt.

Additionally, a static macro cannot be invoked dynamically, e.g. as a result of a call to the **`execute_macro()`** or **`assign_to_key`** primitives. In fact, any callback functions from CRiSP must not be declared as static since the macro will not be in scope.

Static macros are very useful, as they are in the C language, to hide private macros in a large macro so that name conflicts do not occur either with the supplied macros or anything else which has no reason to know the names of these private macros.

Static macros should be used wherever possible to avoid name conflicts which can cause CRiSP to hang or crash by accidentally replacing an internal macro. Conflicts which can occur because the static keyword has not been used can happen at any time especially as CRiSP demand-loads macros.

Modules and static macros

Although the CRiSP macro language looks and feels largely like C, there are inherent problems in try to perform data abstraction and hiding of functions and data. One of the problem areas is to do with static functions. A static function can be referred to within the current file scope, but cannot be referred to from another file. Also, because CRiSP does not provide support for pointers, it can be difficult to hide a static function yet pass a reference to it around.

For example, CRiSP macros make use of registered callbacks, whether for various editing events, key presses, or dialog box events. These events are specified as pure strings, and the CRUNCH compiler does not attempt to parse or understand these strings. The strings are executed at the point they are needed. This is a problem because for most functions it is possible to declare them static and if you get something wrong, such as differing function declarations and definitions, then the compiler will tell you something is up.

It is quite possible to mark a callback static, but not have it in visible scope at the time it is required for execution, resulting in a macro execution error and possibly very difficult to verify correctness of the code.

The modules mechanism is designed to address this problem with run-time support. A module is a way of giving one or more macro files a name such that functions can be identified in these modules. For example, you might implement a macro package to perform searching. The macros to implement these functions may be spread about a number of source files, with many of the macros being private to the implementation, yet some functions are required for external consumption.

With the modules mechanism you can safely make all internal functions static, yet still be allowed to access them in callback functions via the module reference. A module reference is a string such as "search::find_it". The name preceding the "::" characters is the module name, and the "find_it" is simply a macro name, used in much the same way as any other macro reference in a string literal. If no module name is specified, e.g. "::find_it" then this is a shorthand for referring to the current *file*. (CRiSP automatically creates private modules for every loaded macro file. You will see these implicit module references in the debug out from CRiSP if you use the "::function" notation in your code).

Because Because static functions can potentially lead to programming errors at run-time, especially if you have not used the "::" module syntax, then you can use the command line switch "-warnings" which will log entries in the crisp.log file with the keyword "WARNING:" wherever an assign_to_key(), register_macro, register_timer() or create_object(DBOX_CALLBACK) is used which could potentially lead to a run-time error. (Note the -warnings switch is specified when invoking CRiSP, not the macro compiler).

For example, the following is one way of accessing a function:

```
assign_to_key("<Alt-A>", "handle_alt_a");
```

With a module reference you would do:

```
assign_to_key("<Alt-A>", "module_name::handle_alt_a");
```

This facility is designed for implementors of macros who wish to try and maintain as much privacy of the implementation as possible yet need to be able to refer to their own function definitions as part of the entire package. Macro writers should be very careful about calling static functions in someone elses macro package as this negates the whole raison d'etre for static function definitions in the first place.

A module is defined using the *module()* primitive. Refer to the manual page for further details on this function.

Registered Macros

Registered macros are a means of creating a macro which gets called when certain events internal to CRiSP happen. These definitions are available in the crisp.h macro file, and documented in the table below.

Any macro can be registered for one of these events by calling the *register_macro()* primitive. Whenever the event is triggered the macro will be called. A macro can be unregistered by calling the primitive *unregister_macro()*.

An arbitrary number of macros may be registered for the same event. If more than one macro is registered for the same event, then the macros are called in the order they were registered.

Registered macros are convenient short-hand ways of intercepting certain operations within CRiSP.

Type	**Description**
0	(REG_TYPED) This macro is called every time a character is inserted into a buffer via (self_insert).
1	(REG_EDIT) This macro is called whenever (edit_file) is called.
2	(REG_ALT_H) This macro is called whenever the user presses <Alt-H> whilst at the command prompts.
3	(REG_UNASSIGNED) This macro is called if the user presses a key which does not have a macro bound to it.
4	(REG_IDLE) This macro is called whenever the idle timer goes off. The idle timer is set by the CRiSP command line switch (-i). It defaults to 60 seconds.
5	(REG_EXIT) This macro is called when CRiSP is about to exit. It is designed to allow macros to tidy up after themselves (e.g. delete temporary files).
6	(REG_NEW) This macro is called whenever a new file is read into a buffer via (edit_file).
7	(REG_CTRLC) This macro is called whenever the interrupt key is pressed. To avoid confusion, CRiSP sets the interrupt key 'out of the way' to <Ctrl-Y>. (<Ctrl-C> is usually mapped to the center-line-in-window macro). This feature allows macros to be interrupted. Following the usual safe programming style, it is only a good idea to set a flag in the interrupt handler and test its value in the main-line macro.
8	(REG_INVALID) This macro is called if the user types an invalid key at the command prompt. This macro allows the abbreviations and command history feature to be implemented.
9	(REG_INTERNAL) This is called when CRiSP detects a segmentation violation. This can be used to write all buffers away to disk in an emergency. (Refer to core.cr for an example of this macro).
10	(REG_MOUSE) Called when a mouse button is pressed or released.
11	(REG_PROC_INPUT) Called when process input available from a process buffer.
12	(REG_KEYBOARD) Called when keyboard buffer empty. This macro MUST return an integer value. If the value is zero, then keyboard input can be read. If it is non-zero, then the keyboard should not be read, but the internal code will check the push-back buffer for input.
13	(REG_SELECTION) Called when data is available from the PRIMARY selection after the get_selection() macro has been called. This will only happen on windowing systems supporting the ICCCM cut & paste mechanism.
14	(REG_DRAG_N_DROP) This is the drag'n'drop trigger. Currently it is only supported in the XView version of CRISP. When this macro is registered, then the macro will be called when the user drops a file icon into the CRiSP window. The macro is passed a string argument corresponding to the name of the file to be edited.
15	(REG_INPUT_FILENAME) This is the input-filename trigger. It allows a macro to be written which intercepts files being read into a buffer. The macro should return a string value which is the mapped name of the file (or the original filename unchanged). This trigger allows a macro to be written that allows files to be edited which are dynamically uncompressed when read in, for example. See the crisp.cr (infile_trigger macro) for an example of use.
16	(REG_OUTPUT_FILENAME) This is the output-filename trigger. This allows macros to be written which intercept all files which are written to and allows them to perform some file-name specific action. For example, an attempt to write to a symbolic-link could cause a macro to check if the file is a symbolic link and if so remove the symlink and write a

new file.

17	(REG_FILE_MOD) This is the modified file trigger. It is called when Crisp detects that the corresponding file on disk for a buffer has been modified. This is used to detect situations where Crisp hasn't completely read in a file but maybe another editing session has destroyed the file. The calling macro should return an integer value to indicate whether the buffer should continue to be edited, or truncated at the current load point. A value of zero means to stop further editing. A value greater than zero means to continue editing of the buffer.
18	(REG_WINDOW_EVENT) This trigger is reserved for GUI related windowing events. The macro will be called with a string. The current values are: "OPEN" will be passed when the window is mapped to the screen (i.e. uniconised); "CLOSE" when the window is iconised (unmapped).
19	(REG_SCROLLBAR) Reserved.
20	(REG_KEY_ACTIVITY). This is used to detect any keyboard input. It is similar to the REG_TYPED (0) trigger but may be used to detect a key being pressed even if no key is inserted into the buffer. This trigger is only called when CRiSP is waiting in the current process() level input loop. It will not necessarily be triggered by the various primitives which wait for a key (e.g. read_char()).
21	(REG_SCREEN). The current screen has changed, e.g. the user has moved the mouse into another screen window. The macro is called with a single integer argument indicating the screen number which was selected. The calling macro would normally call set_screen() to allow the screen change to take place. If this function is not called then the user cannot change to the designated screen.
22	(REG_PROC_DIED). This triggers calls the specified macro within the context of the buffer which died, indicating that an attached process buffer has terminated.
23	(REG_UNTYPED). This is the exact converse of REG_TYPED, i.e. a trigger will be called after any keystroke which does NOT insert a character.
24	(REG_WRITE_FILE). Called whenever a file is successfully written. Used to allow the audit macro to make a log fo all files which are modified. The registered macro is called with the filename of the buffer which has just successfully been written.
25	(REG_LOCKING). Reserved.
26	(REG_SIGNAL). A SIGUSR1 or SIGUSR2 signal has been received.
27	(REG_EXCEPTION). This callback is used to intercept macro execution errors. The callback macro is passed an error code and a name indicating the macro or variable causing the exception condition.
28	(REG_NEW_FILE_CLASS)
29	(REG_NEW_FILE_INSTANCE)
30	(REG_BOOKMARK) A bookmark has been dropped or deleted.
31	(REG_BUFFER_DELETED) A buffer has been deleted. This trigger is called twice when a buffer is deleted. The first time it is called with an argument which is the buffer-id of the buffer about to be deleted. The second time it is called after the buffer is deleted (buffer id arg is blank or zero).
32	(REG_UNDEFINED_MACRO) An undefined macro was called. Argument passed is name of macro.
33	(REG_INSERT_MODE) Insert mode has changed.
34	(REG_REMEMBER) A keystroke macro is being recorded or has stopped recording.
35	(REG_DELETE_SCREEN) Screen is about to be destroyed.
36	(REG_BUFFER_MOD) Buffer has been modified. This is in contrast to REG_FILE_MOD which indicates something external to CRiSP has modified a file being edited. REG_BUFFER_MOD will happen the first time you insert or delete characters into a clean buffer. This trigger is also called when the buffer is 'unmodified', e.g. if you keep on

	undoing back to the point where the buffer no longer needs saving. You should call the inq_modified() primitive to determine which of the two states you are in.
37	(REG_DEFAULT_FILE). This callback is identical to REG_NEW but is called after REG_NEW and only if the input file does not have an extension. This allows a macro to set up things like colorization type to be written based on the contents of the file, if it cannot be determined from the file extension
38	(REG_CHANGE_DIRECTORY). The current directory has been changed, e.g. by executing the cd() primitive.
39	(REG_STARTUP). This trigger is called after all the startup code has completed, and gives macros a chance to do something before going live. Typically this may be used by pre-loaded macros.
40	(REG_DEFAULT_EXTENSION). Used by the CRiSP macros to intercept editing of files which do not have an extension. The callback macro then attempts to determine the file type.
41	(REG_SELECTION_REQUEST). Used when an external task asks for a copy of the current marked area.
42	(REG_LINE_CHANGED). Callback when a line has been modified. Not officially supported at present.
43.	(REG_CRISP_IPC). Callback to handle special IPC communication under Windows; used to detect multiple instances of CRiSP running.
44.	(REG_PRE_COMMAND). Callback which happens after a keystroke has been typed but before the keystroke is dispatched. Used to implement blinking brackets.
45	(REG_POST_COMMAND) Callback which occurs just after CRiSP processes a keystroke and before going to sleep to await the next event. Used to implement blinking brackets.
46	(REG_BUFFER_MOD2). Called when a buffer is modified. Used to implement file locking.

Timer functions

CRiSP provides a couple of primitives which can be useful for macros which want to perform some activity on a regular time basis. For example, the mail macro allows the user to view mail and uses two functions to watch for mail. When the mail macro isn't actively being used (but has been loaded) then a REG_IDLE registered macro is set up so that every time the idle timer goes off it checks to see if any new mail has been received.

In addition, if you execute the mail macro and are viewing the main mail contents screen, then a timer is setup to poll the mail file to check for new mail whilst you are looking at the current mail file.

This mechanism is implemented using the primitives: *register_timer* and *unregister_timer*. The register_timer() macro takes two arguments - a time value in milliseconds, and the name of a macro to call when the timer expires. When the timer goes off the timer entry is automatically cleared. If you want repeated pulses from a timer then the simplest thing to do is to call the register_timer() macro from the callback function. The register_timer() primitive returns a timer identifier which can be used to cancel the timer via the unregister_macro().

Although the time period is measured in milliseconds there is no guarantee that this level of granularity will be available on the system you are using, and in fact CRiSP may round up your timing request to the next 1 second interval.

The timer may go off some time later than the interval requested, e.g. if a macro is busily performing a lot of computation. Internally, CRiSP checks the timer queues before every keystroke is read, so if you wish to make sure that a timer goes off it may be necessary to force the keyboard to be read.

You should be careful when using timers because you may be called from any environment. For example, you might build a timer to do something at regular time periods. But you will need to consider the fact that the user has popped up a popup window, e.g. the buffer list, or help window and the environment you are in may not be compatable with the operation you wish to perform.

Note that this timer mechanism is independent of the idle timer mechanism described elsewhere. The idle timer mechanism is specifically designed to support the autosave functionality. In theory the autosave mechanism could be built from the timer functionality since it is more generic (although it would be tricky to build an autosave mechanism based on the number of keystrokes typed, which the autosave mechanism does support). The timer mechanism is designed to allow certain classes of macros to poll external events, e.g. files, at a leisurely pace. E.g. it is unadvisable to build a complicated hard real-time mechanism using these primitives because of the inability to guarantee correct operation under all circumstances and operating systems.

Color Support

CRiSP contains support for color. The color support is based on a very simple model which is designed to try and hide some of the peculiarities of running on screens as varied as monochrome serial terminals, to the X11 versions and DOS GUI versions.

The primary primitives for managing colors are the *set_color()* and *get_color()* functions. These functions take a list or return a list of colors. Each element in the list is responsible for coloring a particular part of the screen. E.g. the first element in the list is the normal background color.

Because of the peculiarities of color naming on different systems, CRiSP attempts to offer a facility which can minimise portability problems with user macros when run on different hardware. CRiSP allows you to dynamically select colors for different parts of the screen (using *set_color()*) and also to define a *palette* of colors for certain of the other primitives available.

There are a number of primary display objects which can be allocated colors. Some of these colors are obvious; some are the basis for CRiSPs syntax coloring. The table below lists all the colors that are known by CRiSP.

Code	Mnemonic	Description
0	COL_BACKGROUND	Background color of the screen.
1	COL_FOREGROUND	Foreground color for all windows.
2	COL_SELECTED_WINDOW	Color of selected window title.
3	COL_MESSAGES	Normal color of prompts and messages and Line:/Col: fields.
4	COL_ERRORS	Error message color.
5	COL_HILITE_BACKGROUND	Color of background for a highlighted area.
6	COL_HILITE_FOREGROUND	Color of foreground for a highlighted area.
7	COL_INSERT_CURSOR	Color associated with the insert mode cursor.
8	COL_OVERTYPE_CURSOR	Color associated with the overtype mode cursor
9	COL_BORDERS	Color associated with the window borders.
10	COL_FG_NUMBERS	Color assigned to the foreground of numbers.
11	COL_BG_NUMBERS	Color assigned to the background of numbers.
12	COL_FG_COMMENTS	Foreground color for comments.
13	COL_BG_COMMENTS	Background color for comments.
14	COL_FG_STRINGS	Foreground color for string literals.

15	COL_BG_STRINGS	Background color for string literals.
16	COL_FG_KEYWORDS	Foreground color for language keywords.
17	COL_BG_KEYWORDS	Background color for language keywords.
18	COL_FG_MODIFIED	Foreground color for modified lines.
19	COL_BG_MODIFIED	Background color for modified lines.
20	COL_FG_HASH	Foreground color for directives.
21	COL_BG_HASH	Background color for directives.
22	COL_LINE_NOS	Foreground color for line numbers.

On a non-GUI version of CRiSP the set of color names is built into CRiSP - corresponding to the 16 colors normally available on a serial terminal (e.g. a DOS display using ANSI.SYS or a VT340). On an X11 version of CRiSP you can use any colors available in your systems `rgb.txt` file (as long as you do not run out of pixels in your colormap).

The coloring system is divided into two logical parts -- the colors described above which are normally at the discretion of the user (and setup with the **`Color Setup`** menu) and the color palette. The color palette is a set of colors which may be allocated at any time but which the macro programmer will refer to using color numbers. For example, the *window_color()* primitive may be used to allow each window on the screen to have a background color different from the COL_BACKGROUND parameter described above. Instead of allocating a color by name, colors are referred to using a color value (loosely corresponding to an X11 pixel definition).

CRiSP supports by default 16 colors in the color palette. The macro programmer can define these colors and then refer to them by the color number 0..15.

Colors in the palette and the screen objects are created using the *set_color()* primitive. This function is passed a list of colors corresponding to the colors to allocate. Colors are specified by name and CRiSP will attempt to map the color-strings into the color system supported by the system CRiSP is running on. CRiSP supports a set of 16 standard color names which applications can fall back on if they detect that a non-GUI version of CRiSP is running.

Searching for text -- Regular Expressions

A Regular expression is a term used to describe a string of characters used in pattern matching. Regular expressions allow certain classes of strings to be matched, and provide a flexible way of matching 'token's. CRiSP provides a variety of features when performing pattern matching:

- literal pattern matching.
- character class matching
- wild-card matching
- grouping
- alternation
- repeated expressions
- matching over line boundaries

Many non-alphanumeric characters have a special purpose in a regular expression, indicating a special action to perform. The following table shows the regular expression matching idioms available, with the highest priority at the top of the table. are as follows:

Character	**Meaning**
\x	treat x as a normal character.
@	matches zero or more of the previous expressions.
+	matches one or more of the previous expressions.
{..}	groups a regular expression.
|	performs alternation.

<, %, ^	matches the beginning of a line.
>, $	matches the end of a line.
?	matches any single character.
*****	matches zero or more characters
[..]	matches any character within the [..]
\c	used to set the cursor after a match.
\n	Matches the newline character at the end of a line.
\<	Match beginning of line or non-word character.
\>	Match end of line or non-word character.
	Two regular expressions juxtaposed allow concatenation.

An implied precedence is used with these characters, and it may be necessary to use the '\x' character to avoid certain characters being treated as special characters.

Technically, a regular expression consists of a sequence of one or more simple expressions. A simple expression (SE) is one of the following:

a sequence of characters
< or ^ or %
> or $
[..]
?
*

A simple regular expression (SRE) is a simple expression, optionally followed or enclosed in a modifier:

{SE}
SE@
SE+
SE

A regular expression is a sequence of simple regular expressions as follows:

SRE SRE	(Concatenation)
SRE | SRE	(Alternation)

Character Escaping

The backslash character may be used to precede any character to turn off any special effects the character has. For example to match an asterisk in the text, the sequence "*****" would be used.

A common form of error when writing macros is to forget that the macro compiler strips off the first level of backslash characters. For example, if the user wants to match an asterisk in a macro, he/she might write:

```
search_fwd("\*");
```

However, this is wrong. The macro compiler strips off the '\' and leaves the expression as "*" which matches every line. In order to escape this character properly, the following should be used:

```
search_fwd("\\*");
```

In this case, the macro compiler strips off the first backslash leaving "*****" for the regular expression parser to translate.

The wild card operators: ? and *

The '?' operator matches a single character; '*' matches zero or more characters.

The number of characters matched by a '*' depends on what follows the '*' and the search mode. The **Minimal and Maximal**(pg. 50). matching section describes the issues relating to the length of a matched string.

For example, the following expression:

```
cat*dog
```

matches any line which contains the word cat followed by somewhere else on the line, the word dog.

Character Class: [..] and ..

The square bracket operators are used to match one or more characters from a class of characters. If the expression is of the form '[..]' then a match is successful if the character being matched is any of the characters within the square brackets. If the first character after the '[' is either a '^' or '~', then the match is successful if the character is NOT equal to any of the characters in the matched class.

The characters within the square brackets form either an enumeration or a range of characters. '**[ABC]**' is an example of an enumeration. It matches the single character 'A', or 'B', or 'C'.

'**[a-z]**' is an example of a range. It matches any lower case alphabetic character.

Ranges and enumerations may be combined, for example the following may be used to match a C symbol:

```
[_A-Za-z][_A-Za-z0-9]@
```

which defines a regular expression expression consisting of a single character of '_', an upper or lower case alphabetic, followed by zero or more characters from the class '_', A-Z, a-z or 0-9.

Special characters may be enclosed in the character class construct using the \ syntax. For example, \n matches a new-line; \t matches a tab.

The characters -, and] may be included in the class by preceding them with a backslash (e.g. \- or \]).

The regular expression characters \< and \> can be used as word delimiters. The \< sequence matches either a beginning of line or any non-word character. The \> sequence matches either the end of line or any non-word character. A *word-character* is defined as any of: [A-Za-z0-9_]. These two regular expressions can be used as a short hand way of finding a word without matching the word embedded in a larger word, e.g. **\<begin\>** matches the word *begin* but will not match the word inside *beginning* for example.

Matching Line boundaries

CRiSP allows regular expressions to match text which spans line boundaries. Normally this is not the case. For example, a Unix regular expression of the form: 'a.*b' means match an 'a' followed by any number of characters followed by a 'b'. In this example, the letters 'a' and 'b' are constrained to be on the same line, i.e. the regular expression will not span over multiple lines.

The regular expression sequence '\n' allows a match with the newline at the end of each line to succeed. For example the regular expression: 'fred\nharry' will match the string 'fred' at the end of a line, the newline after fred and the string 'harry' at the beginning of the next line.

The newline matching character can be used inside the character class operator, e.g. [\n] and inside more complicated regular expressions. For example, two match all lines inside the body of a C function can be achieved with a regular expression of the following form:

```
^\{.*\n\(.*\n\)*\}
```

Repetition: @ and +

The @ and **+** are used to indefinitely match a previously specified pattern. A simple regular expression followed by '@' will be matched zero or more times; an SRE followed by '+' will be matched one or more times.

For example, the following regular expression can be used to match a sequence of words followed by a comma (e.g. a sub-phrase of a sentence):

```
{[A-Za-z]+[ ]+}+,
```

[A-Za-z]+ matches any word of one or more alphabetic characters; the [ ]+ matches one or more spaces between each word. The final }+ sequence means repeat the previous expression one or more times.

The following example shows how to match the last word of one sentence and the first word of the following sentence:

```
[A-Za-z]+.[ ]@[A-Z]
```

[A-Za-z]]+ matches the final word in a sentence. The '.' matches the full-stop after it. The expression [ ]@ matches zero or more spaces which may separate the full-stop and the first letter of the next sentence.

Regular Expression Grouping: ..

The regular expression grouping characters are used for one of two purposes - alter the precedence in which the regular expression is parsed, and to define groupings of regular expressions for use by the translation mechanism.

By and large, the regular expressions:

```
xyz  and {xyz}
```

are equivalent. The major use is for bracketing in the presence of the following operators: @, +, and |. For example:

```
{hello}@     {cat}|{dog}
```

The other use for the bracket operators is to define a sub-part of a regular expression for use in translation. Each occurrence of brackets is defined as a grouping. The first occurrence of {..} is group 1, the next is group 2. By grouping parts of a regular expression, translations can be made which swap fields around.

For example, say we have a piece of C code which defines a table as follows:

```
"string1", number1, "string2", number2,    ..
```

If we need to swap the fields around so that we have the numbers first on the line, and the strings following them, then the regular expression search pattern can be defined as:

```
<[ t]@{"[^"]@",}[ t]@{[0-9]+,}
```

This breaks down as follows: <[t]@ matches the spaces and tabs at the beginning of the line. {"[^"]@",} matches the string field (quote followed by zero or more non-quote characters terminated by a quote and a comma). This is the first group. [t]@ matches the zero or more spaces or tabs between the columns. {[0-9]+,} is the second grouping and matches the number followed by a comma.

If the translation replacement pattern (see (translate)) is defined as follows:

```
t1t0
```

then this effects the field swap. The sequence \N where N is in the range 0-9 means insert the matched group designated by N.

Minimal and Maximal Matching

All Unix regular expression parsers use the '*' and '+' operators to mean repeat the previous expression zero or more, or one or more times respectively. CRiSP uses the '@' and '+' operators for the same effect.

However, all Unix parsers, when matching repeated groups will always try to match the longest string. Under Unix, if we have the string:

```
abbbbbbbc
```

and issue the search pattern:

```
b*
```

then this will match the 7 b's between the 'a' and 'c'. By default, CRiSP performs a shortest match. This means that the regular expression:

    b@

will match the zero length sequence of *b*'s starting with the *a*!. For pure searches, the difference hardly ever matters, but when translations are performed the difference is very important. In the above example, using the following translation from 'vi' will result in the following string:

    s/ab*/X/p   Xc

This is what happens with CRiSP:

    translate("ab@", "X")   Xbbbbbbbc

This is simply because the Unix parsers try to match the longest string, whereas CRiSP tries to match the shortest string.

CRiSP provides the ability to modify this default behaviour. This is called *minimal/maximal matching* and backward matching.

The search macros - search_fwd, search_back, translate, search_string and search_list have a parameter, labelled 're' which is used to control the minimal/maximal matching. The minimal/maximal terminology refers to the way that the closure operators (i.e. zero or more or one or more sequences of regular expressions) are matched. This parameter can have one of seven valid values: -3, -2, -1, 0, 1, 2, 3.

The seven case are as follows:

-3	maximal closure, backward
-2	maximal closure, same as search direction
-1	maximal closure, forward
0	forward (literal match)
1	minimal closure, forward
2	minimal closure, same as in search direction
3	minimal closure, backward

0 is used for non-regular expressions. The maximal matching modes are compatible with Unix.

Matching Direction

The difference between *forward* and *backward* matching are that the two mainly comes into effect when performing a backward match. Consider the following line:

```
the cat sat on the mat
_^
```

with the cursor placed on the '*h*' of the first word, '*the*'. In forward matching, if we are searching for the word 'the', then the string 'he cat sat on the mat' will be searched, and the match will be on the word 'the' before 'mat'. In backwards matching mode, the search can start before the cursor, and will match the word 'the' at the start of the line.

Regular Expression Syntax Mode

CRiSP allows the user to select between pure CRiSP regular expression syntax, as described above, or the more familiar Unix syntax. This is done by calling the *re_syntax()* primitive or by setting the SF_UNIX flag to the *re_search()/re_translate()* primitives. In CRiSP mode, regular expressions are exactly as described in the previous sections of this chapter.

In Unix mode, the following features are enabled/disabled:

The '.' character replaces the '?' wild-card character.

The '*' character means zero or more iterations of the previous expression. This disables the CRiSP '@' zero or more character.

The sequence \(..\) replaces the use of {..}. The { and } characters become normal characters.

GUI Based Objects

This section describes the mechanisms and primitives CRiSP provides for accessing such things as scrollbars, dialog boxes, etc. Because different versions of CRiSP may or may not support all of these functions, each primitive can be tested to ensure that the primitive is fully implemented, allowing macros to run in both a pure character-mode based environment or the GUI environment, or both, as appropriate. The macros supplied with CRiSP can operate in both ways, and many of these macros determine their look and feel depending on how they are invoked. E.g. popup dialog boxes are normally implemented in the underlying toolkit (e.g. XView or Motif) when accessed from the menu bar using the mouse, but as character based items when invoked from a keystroke. This means the user can choose between the normally faster keyboard based mechanisms or the more user friendly mechanisms when invoked via the mouse.

The primitives provided for creating graphical objects are designed to complement the macro facilities available. The primitives provide as much functionality as is required, plus a little more. There is a fine dividing line between the functionality available in CRiSP and the functionality available in say, for example, a graphical user-interface tool such as X-Designer or DEVGuide. The idea of CRiSP is to provide high-level functionality because this can help to increase performance whilst keeping the macro interface fairly simple.

One of the design goals of the macro facilities is that macros should be as portable as possible, i.e. the intricacies and idiosyncrasies of particular GUI toolkits are hidden. This means it is easier to program the macros, but at the risk of losing absolute functionality. Some aspects of the GUI objects expose the underlying toolkits but only in a manner which is portable. (For instance, the push-pins in the OPENLook interface specification are accessible to the macro programmer whether CRiSP is running under an OPENLook compliant toolkit or not).

One of the main aims of the macro interface is to put as little policy and look and feel into the source code of CRiSP. This means that the user interface and any stylistic issues can be modified by changing the supplied macros. This means that CRiSP is more reliable since small minor changes are not needed to change trivial aspects of the user interface. It also means that many of the look and feel issues can be modified across the platforms on which CRiSP is supported simply by changing the macros.

Terminology

It is necessary to define some terms which are used in reference to the GUI objects which can be created.

attribute

A particular object has one or more values. These values can range from the *value* of a user interface component to an intrinsic property of that object. For example, a list box has a value corresponding to the currently selected item, but has multiple attributes which control its size, and cursor positioning.

dialog box

A dialog box is a container for the objects contained within it.

object

The term *object* is used to refer to a displayable graphical item (more commonly referred to as a *widget* in X11 parlance, or a child control in Microsoft Windows). CRiSP GUI objects do not necessarily map identically to widgets in the underlying implementation. The CRiSP GUI objects are designed to be high level enough to allow the invoking macros to get the job done very simply, yet providing the power and hooks for more advanced usage. *objects* refer generally to things such as scrollbars, menus and dialog boxes. Do not confuse the use of the term *object* with object oriented programming concepts. The term object is used to describe an abstract or indeterminate data type.

Windowing Systems Programming

The CRiSP dialog box and object support is something that has grown during the course of numerous versions of CRiSP. The original aspirations of the dialog box mechanism was to allow the creation of familiar dialog boxes for the user interface as CRiSP evolved from a purely character based piece of software to a GUI based product.

The initial idea was to allow generic but rigid dialog boxes to be supported. The very first incarnations was to support the following:

1. Open File dialog.
2. Save File dialog.
3. Search & Translate dialog.
4. Font selection dialog.
5. Color configuration dialog.

The very first implementations of these were rigid and hard coded within the CRiSP software. It became very obvious shortly thereafter that serious platform specific concerns would be necessary, for example a File Open dialog box which did not support multiple drive selection on a Windows platform would be little more than useless. Also it was clear that the various controls available in a dialog box would need to be tweaked or customised depending on a large number of factors. This rigid structure paved the way for something more generic.

The primary idea of the dialog box system is to provide user interface components which can be used to enhance and customize the CRiSP software, yet achieve a native look & feel.

GUI software development is one of the most challenging aspects of software production and over the years, such organisations as Microsoft, IBM, and the OSF have strived to achieve the ultimate user interface. None of these organisations has produced a *finished* product, in the sense that no matter what they do, there are always more tweaks and improvements which can be made.

Dialog boxes and the objects contained within them are fairly abstract. A dialog box is like a box containing numerous objects inside it, and your job as a macro programmer is to come up with an aesthetic and obvious user interface handling the events which the user invokes as a result of clicking on objects or typing into them.

The dialog box architecture is loosely based on concepts and ideas which come from: the XView API and the Xt toolkit API. The XView API, which has now been abandoned by Sun Microsystems is a simple and easy to use API which allows for the creation of user interface components using a fairly simple syntax and style. The XView API suffers because it has limited scope for object orientation in its real sense, and it is difficult to create nested components. On the other hand, the Xt API is a very large and complicated API which can be very difficult to understand.

The *de facto* industry standard, the Microsoft Windows API is also a complex API with hundreds of functions.

One of the problems in building a user interface library is to provide simplicity of operation combined with enough expressive power to achieve anything possible. No commercial API has achieved that because, quite simply, it is not possible to support a user interface with wildly differing requirements but at the same time to achieve programming simplicity. User interface programming is more akin to real-time programming where things can happen at random, depending on the actions of the user.

CRiSP does not attempt to solve all problems. It simply provides a programming paradigm that is sufficient for its purposes of allowing the user interface of CRiSP to be extended and customizable.

In attempting to learn and understand the CRiSP API, it is useful to have some grounding in windowing system API programming. It is important to understand the issues before coding anything. GUI programming has proved to be a difficult area for the industry and there are numerous reasons why.

When programming something like a dialog box, it is typical that the functional requirements will change. A layout or user interface component may need enhancing. A program structure which appeared reasonable to start off with can be a maintenance nightmare as the program evolves.

Consider the following. In a typical Microsoft Windows environment, a dialog box is created by a description. This description can come from a plain text file (a resource file) or from an interactive tool, such as the Visual-C appWizard. Both of these approaches are fine for quickly creating a dialog box, but they can be very tortuous to change a dialog box, or to have it adapt. For example, a dialog box may contain a single line input field and an OK button. What happens if the user wants a different sized font? Microsoft Windows until very recently has not addressed this issue. If you increase your screen resolution to get more pixels on the screen then your dialog boxes get proportionately smaller until at some point they are difficult to read.

On the other hand, the X11 windowing system addresses this area very cleanly, by virtue of 'geometry management'. Geometry management is a very important concept which is very difficult to understand completely. In the Microsoft Windows environment, you tend to describe dialog boxes by using absolute pixel co-ordinates. This means you cannot scale things very well because otherwise one user interface component can extend into and overlap another one. With the X11 system (as promoted by the Xt and Motif toolkits), dialog boxes tend to be described by a set of layout *semantics*. Instead of saying "Input field#1" is located at x,y co-ordinates (100,200), and that the "OK" button is located at (200, 250), you would say, "place the OK button below the input field". Now if the user were to use a larger or smaller font, the relative placement of the objects will avoid any form of overlap and maintain an aesthetic appearance.

Of course, this description is overly simplistic. You can achieve either effect with either windowing system. However, the code to do this is not necessarily obvious. It is not uncommon with GUI programming to spend inordinates amounts of time solving seemingly trivial problems, such as why doesn't this push-button appear where I expect it to.

At best it is safe to describe windowing programming as a non-linear programming methodology. This means that you do not follow a sequence of steps and eventually obtain the desired end result. Instead you handle multiple lines of execution at the same time, and you try to prove your code is correct by repeated testing.

Overview of the Dialog box system

In order to understand how CRiSP supports dialog boxes, it is necessary to think of the steps involved in creating a dialog box.

1. Firstly, we sketch or conjure up a mental model of what the dialog box is going to look like. E.g. we consider the user interface components, such as input fields, toggles, push buttons. Most dialog boxes have a common core of functionality, such as an OK and CANCEL button. Stylistics issues dictate that these are normally at the bottom or right hand side of a dialog box.
2. Next we look at each user interface object and decide what its purpose is, how it responds to user input.
3. We then sketch up a prototype showing visually what the final dialog box will look like, without any semantic code to implement any actions.
4. Next we write the code for each object within the dialog box in turn.
5. We can then look at the dialog box as a whole and test it so that there are no unnecessary interactions between the individual components. (E.g. we may decide that the <Enter> key is used to insert a line of text in our input field, but this may conflict with <Enter> being used as the default action for the OK push button).
6. Finally we integrate the dialog box with the rest of CRiSP, maybe adding a new menu option or icon to invoke the dialog box.

What started out as a simple idea can lead to a lot of coding and trial and error until the dialog box finally performs as we like. Sometimes, we may find limitations. For example, some semantic action may not be achievable and the overhead of coding for certain actions may be so large compared to the rest of the coding that it may be a matter of accepting what is achievable rather than striving for perfection. This is a common way of coding and is responsible for glitches in the workings of many user interfaces.

Now lets consider these steps again. The first part of sketching a user interface may seem trivial but the more effort put in at this stage then the easier it will be to come up with the final result and avoid huge amounts of debugging.

The next stage is to try and come up with a user interface prototype of the dialog box. This is a fairly mechanical process with a certain amount of trial and error. It can be very daunting trying to understand why there are so many objects and attributes and which can be used together. Until you understand the rationale for these attributes you will be programming in the dark, and will end up resorting to trial and error. This is a fact of life for many newcomers to windowing systems programming. It is simply not possible to absorb the thousands of pages of documentation which accompany commercial window systems, and it can take many months or years to reach any level of competence.

Coding of the semantics for each user interface component is fairly straightforward but this is where all the errors in the user interface will creep in. We simply try out each object in turn, clicking, dragging, selecting, etc and putting in the code to handle each event.

The later parts of the testing will involve refinement of the coding stage as conflicts and changes in ideas surface.

Overview of an Object

A CRiSP *object* is equal in status to a Windows child-control or an X11 *widget*. It is a name to refer to a type of user interface object, which responds to some user input and can issue callbacks. An object is an entire subsystem in itself. If you look at CRiSP buffers, or windows, you find that there are ways to create the objects, and various functions for retrieving and setting attributes. Objects are similar in this respect, but instead of having hundreds of primitives which control the way an object works, we refer to attribute values.

An object is a reasonable term to use, since the user is not concerned about the implementation of the object, and a certain amount of class inheritance is used internally to provide a more consistent way of handling and looking at the objects. Each object has numerous attributes which control its behaviour; some of these are specific to the actual type of the object, e.g. the bitmap used in an icon, whereas other attributes are generic to all the objects, e.g. the (x,y) co-ordinates of the object.

Objects can be considered to be implemented from two parts: a part called the *core* part (similar terminology to the Xt library), and an object specific handler. By thinking of objects in this way, it is a lot easier to understand where to look for specific documentation on the attributes available for each object. In terms of the actual implementation, this means a lot of common code is shared amongst the objects, and in terms of the documentation, it means there is no need to repeatedly document the common attributes to all the object types.

At the most basic level, you can think of a dialog box as an *umbrella* containing a collection of objects inside it. At various times you need to refer to the individual objects within the dialog box, e.g. so that you can retrieve the current status of the object, change an attribute value, or decide in the callback code, which object has just been actioned on by the user. In order to do this, you need to give an object a *name* (via the DBOX_NAME attribute). It is not mandatory that you do give an object a name, but you will not be able to refer to the object without one. (You might create DBOX_SPACER objects in a dialog box to control the spacing between user visible objects, but a DBOX_SPACER object has no semantics so there is little point in giving an object a name).

A name is an arbitrary string used as an argument to the DBOX_NAME attribute. There are no particular restrictions on the syntax of the name, but it is useful to something meaningful. Every object inside a dialog box should have a unique name or no name. If two or more objects have the same name, then it is indeterminate as to which one will be accessed when you refer to an object in your macro code.

Different dialog boxes can contain objects with the same names as objects in other dialog boxes. Effectively the name of an object within a dialog box can be considered to be of the form: <obj_id>.<name_id>, where <obj_id> is the return value from the create_object() primitive, and the <name_id> is the value of the DBOX_NAME attribute.

Remember that CRUNCH supports *switch* statements using string expressions. If you look at the supplied CRiSP macros you will find that switch statements are used in many of the dialog box handling macros to distinguish the object that caused the callback to occur.

Creating a Dialog Box

A dialog box is created via the *create_object()* primitive. The create_object() primitive takes a single argument, which is a list describing the dialog box. This may sound simple, but real dialog box descriptions can extend over hundreds of lines.

The first thing is to break down the description of a dialog box into its components: push-buttons, input fields, etc. CRiSP supports a number of user interface components.

Before progressing any further, let us show a very simple dialog box description:

```
int obj_id;

obj_id = create_object(make_list(
        DBOX_TITLE, "Title at the top of the dialog",
        DBOX_CALLBACK, "callback_function",

        DBOX_LABEL, "This is some text in the dialog box",

        DBOX_BUTTON, " Ok ",
                DBOX_DEFAULT_BUTTON
        ));
```

This dialog box is extremely trivial but illustrates a number of basic areas of the dialog box creation mechanism.

The first thing to notice is the declaration of an integer variable, *obj_id*. When a dialog box is created, an object identifier is returned which allows the dialog box to subsequently be referred to, e.g. for the purposes of inquiring about attribute values, or for changing attribute values.

The next thing to notice is the use of the *make_list*() primitive which is used to create a list argument which is passed to create_object(). Remember that create_object() takes a single argument. The reason for this, is that a dialog box can be conditionally constructed by creating a list, and then calling create_object() with that list parameter. If create_object() took multiple arguments, then it would be very difficult to support conditional arguments. (The C language does not support conditional function arguments either). The actual list argument contains elements which are integers, strings or nested lists of arbitrary complexity.

When a dialog box is created, you can picture the argument list definition like this:

```
attributes of the dialog box itself
object-1
        attributes for object-1
        ....
object-2
        attributes for object-2
        ....
        ....
object-n
        attributes for object-n
        ....
```

You will notice in the supplied macros that indentation is used to show this implied nesting structure.

There are various attributes used which affect the dialog box itself, including a title to appear at the top of the dialog box (DBOX_TITLE), a single callback routine which is used to handle all events on the objects within the dialog box (DBOX_CALLBACK), and other attributes, such as allowing the dialog box to be resized or not.

All attributes start with the prefix **DBOX_**. All these attributes are listed in the <include/gui.h> include file supplied in the CRiSP source directory. Some of these attribute definitions are there purely for historical reasons: either they do not work, or are incomplete in various platforms. The ones you can rely on are properly documented, so you should be wary about trying to deduce the functionality of an attribute based on what you see in the include file.

As mentioned, all attributes start with the DBOX_ prefix. All user interface objects also start with the DBOX_ attribute as well. (This is not a good thing as it can be difficult to determine whether an element of the list starts a new object definition or is an attribute definition for the preceding object, however for historical reasons the naming convention remains).

Within the <include/gui.h> file are a list of the object types which are defined. Again, some of these are remnants of history and are not actually implemented.

Object Types

The following table lists the objects which are defined.

Mnemonic	Description
DBOX_BUTTON	This implements a push button, e.g. typically used for OK or CANCEL buttons.
DBOX_CHILD	This object type is never explicitly created but is used internally for object grouping. You should never refer to this object type as attempting to do so is undefined.
DBOX_COLOR_SELECTOR	The color selector object type implements an object which is used within the Color Setup dialog box to support the R-G-B slider controls. This object type is very high level and very specific to the job of implementing that particular dialog box. It is implemented as an internal composite object to relieve the color setup macro from having to implement numerous semantic actions for the various sub-components.
DBOX_COMBO_FIELD	A DBOX_COMBO_FIELD is very similar to a DBOX_FIELD, in that it displays a label and an input field to the right of the label. In addition, a drop-down list of available options is available.
DBOX_CONTAINER	A container object is similar to the DBOX_GROUP_START and DBOX_GROUP_END object types and is used to create a sub-dialog box.
DBOX_CORE	This object type is never created or referred to explicitly. It is used as part of the internal object

	oriented implementation.
DBOX_DIRECTORY	An list type object which contains semantics for implementing a selection from a set of valid files or directories.
DBOX_DISPLAY	This object type is never explicitly created in a create_object() definition, but is used to refer to the physical screen. For example, using this object type it is possible to inquire about the physical screen size.
DBOX_DRIVE	(Windows only). Used to implement the drive-selection combo field as seen in the contents window file selector, or the change directory dialog box.
DBOX_DROP_SITE	A drop-site is a special object type used to create a drag and drop site. It currently only has a meaning for the XView implementation of CRiSP as that is the only windowing system which supports a visible drop site.
DBOX_FIELD	This creates an input field. An input field consists of a label to the left of an input field. The actual appearance of an input field is GUI system dependent.
DBOX_GROUP_END	A DBOX_GROUP_END object is used like a close parenthesis to terminate the definition of a sub-dialog box.
DBOX_GROUP_START	A DBOX_GROUP_START object is used like an open parenthesis to create a sub-dialog box.
DBOX_LABEL	This object type creates a static label of text. The object type can display a single line of text; no interpretation of newline sequences or other escape codes is supported. If you need multiline text then multiple labels should be used.
DBOX_LINE	A simple object used for drawing 3D lines within a dialog box.
DBOX_LIST	This creates a multiline scrolling list box.
DBOX_MENU	A DBOX_MENU is used to implement a popup menu.
DBOX_MENU_BAR	The menu bar object implements the menu typically seen at the top of the main CRiSP edit window. It has no semantic actions itself but is used as an anchor for the individual DBOX_MENU_BUTTON objects created.
DBOX_MENU_BUTTON	A menu-button is used to define the root of a menu tree associated with a DBOX_MENU_BAR.
DBOX_MENU_ITEM	Implements a selectable menu item.
DBOX_MENU_SEPARATOR	Used in menu creation to create a line or blank space separating menu items.
DBOX_OPTION_BUTTON	A special type of button which allows the user to select from a list of valid alternatives.
DBOX_OWNER	This object type is never explicitly created in a create_object() definition, but is used when changing attributes to refer to the dialog box itself.

DBOX_PANED_WINDOW	A container object used to contain two child objects. The real-estate allocated to the children can be changed by using a slider located between the windows. Used in CRiSP to implement the contents window.
DBOX_RTF	An object which understand the RTF (Rich Text Format) text format and is used as the basis of the hypertext help system.
DBOX_RULER	Ruler which is associated with a CRiSP edit window.
DBOX_SCREEN	A screen object corresponds to a CRiSP character mode editing screen. (Remember that a CRiSP screen object can contain multiple character mode windows). A DBOX_SCREEN object on its own doesn't do very much, but in combination with other primitives the DBOX_SCREEN object is responsible for the main CRiSP user interface.
DBOX_SCROLLBAR	Implements a scrollbar.
DBOX_SLIDER	Reserved for future implementation.
DBOX_SPACER	A spacer object has no display or input semantics, but is used to force spacing between the other objects, i.e. it occupies physical dialog box space, but appears to be a part of the background separating the other objects.
DBOX_STATUS_BAR	Used as the parent object for individual DBOX_STATUS_PANEL objects.
DBOX_STATUS_PANEL	Implements a separate text pane for the status panel.
DBOX_SUB_GROUP_START	Similar to DBOX_GROUP_START.
DBOX_TABBED	Obsolete.
DBOX_TABBED_WINDOW	A container widget which allows tab selections to be placed at the top of the object area for controlling visibility of the underlying children.
DBOX_TABLE	Implements a spread-sheet like object containing individual rows and columns.
DBOX_TEXT	This creates an object similar to a DBOX_LIST, but no semantics are associated with the object, e.g. no callbacks are generated when attempting to click or edit the contents of the box. This object type is one of the oldest types implemented and was designed to support multiline scrollable data. It has now passed into obscurity as many of the other types offer more functionality.
DBOX_TOGGLE	A toggle object is used to implement an array of 1-32 toggle buttons. These buttons can have exclusive or non-exclusive behaviour (e.g. like Microsoft Windows radio buttons).
DBOX_TOOL_BAR	A tool bar object acts as the root for individual tool buttons placed inside it.
DBOX_TOOL_BUTTON	A tool button is the actual implementation of a clickable icon button.

DBOX_TREE	A tree-like widget which supports outlining views of data. A tree widget is a bit like a DBOX_LIST widget but you can control the visibility of hierarchies of the data.
DBOX_VSCROLLBAR	Obsolete.
DBOX_HSCROLLBAR	Obsolete.

Some of the object types described above are not fully implemented. They may be in future versions, but there is no guarantee of this.

For a long period of time, CRiSP has evolved internally to create the object structure described in this manual. Some of the mechanisms are object types are artefacts of this historical evolution, but now that CRiSP has become a stable and mature software product, it is unlikely that many of these historical artefacts will be removed. These artefacts will be documented as necessary.

Geometry Layout

The dialog box mechanism of CRiSP provides for various *geometry management* mechanisms. Geometry management is a fancy way of saying how objects are to be positioned in a dialog box.

In the early versions of CRiSP, the layout mechanisms were very simple and these simple ways evolved to offer a lot of flexibility. Geometry management is a complicated area and it is this area in which you can waste a lot of time trying to lay things out properly and understand why things do not appear the way you expect them to be. Geometry management is complex because there are so many different requirements for the laying out of objects within a dialog box. For example, you may want a particular input field to be so many characters wide. Another input field may be described as being stretchable, so that if the dialog box is resized by the user, then the input field expands to make use of the extra space. Another possibility is to constrain the width of one object to be exactly the same width as another object, e.g. an input field place directly above or below an input list.

There are no hard and fast rules about geometry management, and it is always advisable to try one thing at a time. If you try to constrain the objects in a dialog box too much, then you may end up producing inconsistent constraints leading to peculiar behaviour.

It is worth describing the evolution of the geometry management mechanism so that you can understand the need for complexity.

Every object created inside a dialog box can be treated as a rectangle occupying a piece of the physical area of the dialog. Under default conditions, the objects created inside a dialog box are placed immediately below the preceding object. On its own this would lead to some very restricted and horrible looking skinny dialog boxes. To this model was added the ability to place objects next to each other. (This is done with the DBOX_NEXT_COLUMN attribute). Thus, using the default rules we can create a dialog box containing a series of objects, some next to each other and some below each other.

These initial semantics are actually fine for most dialog boxes, but they do not provide a very fine level of control. One issue that must be met head on is the issue of the different platforms. For example, a push button under Microsoft Windows has a very different appearance and size characteristics compared to a Motif or XView button. It is therefore important to avoid trying to lay things out in a dialog box using absolute pixel values as this would be fairly disastrous, if contemplating using your dialog box on a different system. (Even using a dialog box by a different user can lead to problems as each user may have a different default font, and the size of a push button or label may be different depending on the font actually used).

It is therefore wise to treat each object as a rectangle for which you do not know its eventual size. This leads to problems, for which various solutions are described below.

Now we can layout a series of rows of objects. The next thing to be implemented was support for equi-spaced push buttons, e.g. like those normally located at the bottom of the dialog boxes (e.g. OK, Help, Cancel). This is impossible to do with a simple policy of laying one rectangle out on the same row as another or below a row of rectangles. To this was added the DBOX_CENTERED attribute. When this attribute is used, then the object is centered within the free space of a row. For example, you will see typical usage of this attribute for the buttons at the bottom of a dialog box, e.g.:

```
        obj_id = create_object(make_list(
                ....
                ....
                DBOX_BUTTON, " Ok ",
                        DBOX_CENTERED,

                DBOX_BUTTON, "  Apply ",
                        DBOX_NEXT_COLUMN, TRUE,
                        DBOX_CENTERED,

                DBOX_BUTTON, " Help ",
                        DBOX_NEXT_COLUMN, TRUE,
                        DBOX_CENTERED,

                DBOX_BUTTON, " Cancel ",
                        DBOX_NEXT_COLUMN, TRUE,
                        DBOX_CENTERED
                ));
```

These very basic layout semantics are now sufficient to describe most types of dialog boxes, but there is still a long way to go.

One of the next issues to resolve is what happens when a dialog box is resized? Many dialog boxes are fairly static in nature, and having them resizable is of little benefit. With the attributes described above, nothing has been said of resizing a dialog box. Without any extra information CRiSP cannot determine what to do, and the end result of the user making a dialog box bigger is simply to add more blank space at the bottom and to the right of the dialog box. (The exception is those objects marked with the DBOX_CENTERED attribute which will scale the gap between the objects to make use of the extra space).

Resizing dialog boxes, and Constraint management

For some dialog boxes, it is plainly difficult to decide what is a reasonable size for the objects inside them. For example, the buffer list which CRiSP will display comes up by default with about 8 lines for the file list. If you have less than 8 files loaded into CRiSP then some of the space used by the buffer list is wasted. On the other hand, if too much space is used then it is conceivable that either the user will not be able to *see* the entirety of the dialog box (e.g. on a low resolution screen), or the dialog box will simply obscure other windows on the desktop and get in the way.

So now you not only have to decide how to lay out the objects in your dialog box, but you also need to consider if you want to allow the dialog box to be resizable and if so, how the dialog box should stretch or shrink depending on the user's actions.

In general, there are a number of clear cases for stretchable objects within a dialog box. When we talk about stretchability we are referring to objects whose rectangular shape can be expanded or contracted by increasing the height or width of the rectangle enclosing the object. At one extreme, we have objects which are unlikely to be stretched, for example a text label would normally not expand or contract as a result of a dialog box being resized; a push-button usually doesn't change it's size or shape, and neither would an individual icon in a toolbar.

When looking at a dialog box, you can normally distinguish two types of objects within the dialog box: the major ones providing the raw information, and the *noise* objects which are there simply to provide a means for actioning events, e.g. the push-buttons. It is the main objects which are important and the reason why a user would want to resize a dialog box. One way to handle resizing is to allow these major objects to scale with the size of the dialog box, making use of any extra space the user allocates and contracting. The main CRiSP editing window illustrates this very clearly. The main editing area expands to fill the space, and when you resize the CRiSP window, you are doing it so you can see more of the editing area. You wouldn't normally resize the CRiSP window in order to get a fatter status bar or menu bar.

So an attribute is needed to mark an object as being able to expand to fill the space. In the early versions of CRiSP, this was provided by four attributes: DBOX_ANCHOR_TOP, DBOX_ANCHOR_BOTTOM, DBOX_ANCHOR_LEFT, DBOX_ANCHOR_RIGHT. These attributes indicate that the object should expand to touch the sides of the dialog box and when the dialog box is resized to maintain the relationship. An object marked with DBOX_ANCHOR_LEFT and DBOX_ANCHOR_RIGHT would typically be used for something like the list box in the buffer list or the main CRiSP editing window so that the object extends from one side of the dialog box to the other, exactly filling the space. Likewise, specifying DBOX_ANCHOR_TOP and DBOX_ANCHOR_BOTTOM would enable an object to expand to fill the height of a dialog box.

When laying out a dialog box, CRiSP tries hard to avoid objects from overlapping, so it is normally safe to use the DBOX_ANCHOR attributes even if there are objects to the left, right, top or bottom of the specified object.

With what has been described so far, it is possible to describe fairly abstract dialog boxes where we do not care too much about the fine level of detail in a dialog box, and thus we can be reasonably user and environment independent.

One of the major missing features from the discussion so far is a fine degree of granularity. For example, consider a case where we want to have two list boxes side by side. This is easily supported in CRiSP and it is easy to ensure that the list boxes are allocated 50% of the space each. (It is not possible to specify that one list box has 25% of the space and another 75% of the space). Now consider putting a label object or an input field above each of the list boxes. The width and characteristics of a label are not the same as a list, and without hardcoding exact pixel widths it is difficult to ensure that the labels or fields are exactly the same widths as the list boxes beneath them, especially after a resize operation.

In order to handle this type of fine level layout, we need to implement a mechanism for constraint management. Constraint management is a way of giving hints to say things like: "Object X is to be exactly as wide as Object Y", or "Make Object X and Object Y align with each other on the bottom side" (e.g. placing a label field aligned to the left and to the bottom of a multiline list box).

Constraint definitions is a very powerful area but it is exceptionally easy to create constraint definitions which are impossible to satisfy leading to incorrect looking displays. Worse, it is very difficult for the human mind to understand why the layout is not as you expect. Therefore, when using constraints, you should be very careful and try one thing at a time.

The constraint mechanism with CRiSP is loosely based around the **Form** widget available in the Motif programming API. The Form widget implements a mechanism for stretchability, which allows an objects size and position to be tied to any other object in the dialog box. The way this works involves a number of attributes, but essentially works like this. When you create an object you can tell CRiSP that the top, bottom, left or right hand side of the object is *attached* to the top, bottom, left or right of some other object in the dialog box. For example, lets take the case of a single line input field, which we want to place above a multiline list box. If we want the input field to be exactly the same width as the list box, then what we want to say is something like this: the left hand side of the input field has the same X co-ordinate as the left hand side of the multiline list box, and also at the same time the right hand side of the input field has the same X co-ordinate as the right hand side of the list box.

In order for this to work properly, in this case, we would also need to tell CRiSP that the input field is stretchable. (We do this with the DBOX_ALLOW_RESIZE attribute). If we did not make the input field stretchable, then it is a bit like try to stretch a wooden ruler - you cannot, and you end up with something which is not as you expected. Sometimes it is reasonable for an object to be stretchable, sometimes it is not. Normally, stretchability is important when you want the objects within the dialog box to act like pieces or rubber. An example where stretchability is normally inappropriate is for buttons, but these are catered for using the DBOX_CENTERED attribute as described in the previous section.

Now the complexity comes because you now have 4 sides to your object which can be combined to refer to the 4 sides of any other object, or even any 4 distinct objects. Therefore, you have enough rope to hang yourself, and you should handle constraint properties very carefully as it is very easy to specify impossible conditions, such as constraining the layout of an object to itself.

As well as being able to constrain an object so that its size or position is with respect to some other object, you can also constrain an object so that it maintains some fixed distance from the edges of the dialog box. This in effect gives a similar effect to that previously described for the DBOX_ANCHOR_xxx attributes described, but is slightly more functional.

The following example illustrates some of these points. Consider the case of a dialog box, which has an input field, beneath which is a list box. The input field is constrained to be the same width as the list box, and beneath the list box is an Ok and a Cancel button:

```
obj_id = create_object(make_list(
        /* Give a title to the dialog box. */
        DBOX_TITLE, "Constraints Example",


        /* Macro to callback when user does things */
        /* in the dialog box. */
        DBOX_CALLBACK, "callback_macro",
        /* The first object is an input field, */
        /* and we have a label to the left of the */
        /* input field. */
        DBOX_FIELD, "Please type in: ",
/* The input field is marked as stretchable */
                DBOX_ALLOW_RESIZE, TRUE,

                /* Constrain the left side of the */
                /* input field to be in the position as */
                /* the left side of the object called */
                /* "list" */
                DBOX_ATTACH_LEFT_TO_LEFT, "list",

                /* Do the same for the right side */
                DBOX_ATTACH_RIGHT_TO_RIGHT, "list",

/* Create a scrolling list widget with 3 lines in it. */
        DBOX_LIST, quote_list(
                "line 1",
                "line 2",
                "line 3"),
                DBOX_NAME, "list",

        /* Create a push button so we can dismiss */
        /* the dialog. */
        DBOX_BUTTON, " Ok "
        ));
```

A more complete description of constraints and the attribute values is described in the manual page entries for the attributes.

Object hierarchies: menu bars, tool bars, status panels

When building a dialog box layout, what you are really building is a *tree* of objects whose grand parent (or **root**) object is the dialog box owner (referred to as the `DBOX_OWNER` object). For a simple dialog box, you can simply layout objects in multiple rows, with each object adjacent to the previous one, as described previously.

The tree-of-objects idea comes into play in two contexts: certain objects act as place-holders, or umbrellas for multiple constituent sub-objects; also, you can create a tree of objects to refine the layout of a dialog box.

In the first case, placeholders are used for certain types of objects. The three most notable objects are for the menu bar, toolbar and status panel. In each of these three cases, we have a generic object type, e.g. the menu bar, but there is a lot of fine level of detail that is needed to describe each menu entry, or each icon. The mechanisms for supporting these three types is slightly steeped in the history of evolution of CRiSP, and so each is subtly different from the other, but the idea is the same.

The idea of the objects is to define a rectangle of area within which the object is displayed and can respond to user actions (mouse and/or keyboard events). To make life easier for the macro programmer, a lot of housekeeping is hidden behind the scenes and a relatively simple interface is provided. If we take the menu bar, for example. A menu normally lives at the top of a dialog box and is as wide as the dialog box, even if this means that some menu entries are not visible because the owning dialog box is too narrow. The point of interest here, initially, is the real-estate space taken up by the menu bar, and not the real-estate taken up by the individual menus and menu items. (Especially as these are invisible until the associated menu button is clicked on). Mentally, we think of a menu bar and the associated menus and sub-menus as a hierarchy. When programming them in a CRiSP macro, you will lay them out in a hierarchy.

As described before, you create a dialog box containing a list of objects to be displayed. In the case of the menu bar, toolbar, and status bar, you create a *composite* object. This object acts like a single object, but contains a tree of sub-objects. The sub-objects are restricted in their object types. For example, you only create `DBOX_MENU_BUTTON`s as *children* of the `DBOX_MENU_BAR`, and `DBOX_TOOL_BUTTON`s are created as children of a `DBOX_TOOL_BAR`.

For the DBOX_MENU_BAR case, only DBOX_MENU_BUTTON or DBOX_DROP_SITE types are reasonable children. For DBOX_TOOL_BAR, only DBOX_TOOL_BUTTONs are valid. For the DBOX_STATUS_BAR, DBOX_STATUS_PANEL are the sub-objects which can be used.

In the case of the menu bar, it is necessary to define the hierarchy of sub-menus and cascaded menus which can be created, along with any attributes each of the sub-objects may require. Toolbars are flat data structures, in that there are no hidden objects which suddenly popup when activated, but there are numerous options and attributes which can be specified, many of which are common to all the buttons in the toolbar. In the case of a status panel, it is useful to be able to split the visible area of the status panel in such a way so that multiple areas of text can be displayed, and also to allow some degree of rubberness, e.g. the standard status panel which is displayed when CRiSP comes up has a flexible status message area to the left, whilst the sizes of the INS/Line:/Col: fields are relatively static and do not take up any slack when the main window is resized.

Object hierarchies and grouping

As described in the previous section, it is sometimes necessary to create a sub-tree of objects, either because it is easier to treat a group of objects with common functionality, or because of layout semantics. In the various descriptions of objects and attributes described so far, it has been shown how to treat the layout of a dialog box by treating the objects as rectangles which are laid out next to each other or below each other. There is a certain class of layouts which are difficult or impossible to do with these mechanisms, or with the constraints mechanisms.

Consider the layout of a dialog box which consists of a small label, centered above a multiline list box. This sort of layout is easily describable, with something like:

```
obj_id = create_object(make_list(
        ...
        DBOX_LABEL, "Items",
                DBOX_CENTERED,

        DBOX_LIST, quote_list(....)
        ...));
```

Now consider two lists, side by side, with a label above each one. One way to do this is with something like:

```
obj_id = create_object(make_list(
        ...
        DBOX_LABEL, "Items#1",
                DBOX_CENTERED,
        DBOX_LABEL, "Items#2",
                DBOX_CENTERED,
                DBOX_NEXT_COLUMN, TRUE,

        DBOX_LIST, quote_list(....),
        DBOX_LIST, quote_list(....),
                DBOX_NEXT_COLUMN, TRUE,
        ...));
```

The problem is that this is unlikely to work. We need to lay out the objects in approximately the left to right, top to bottom approach. The effect of the above is that the labels will be centered within their own 50% of the row, but the relationship of the labels to the list boxes will not necessarily be what you want. For example, if we extended the dialog box description above to include an object, say a push-button, to the right of the second DBOX_LIST, then the two labels would no longer be centered directly above the corresponding list boxes.

The starting point for this type of layout was very simple: take something simple that works, and create two copies of them side by side. But the problem is that we were not precise enough in our original description and this lack of exactness shows through as soon as we try to modify the description to encompass other objects in the dialog box.

This sort of thing is very frustrating and is why it is always a good idea to build up a dialog box a piece at a time - at the point a dialog box description breaks, you can backtrack and try and understand what has gone wrong. Trying to understand the problems with the description above can be very difficult if it is a complex dialog box.

There is a solution to this, and it is called *object-grouping*. Object-grouping is a mechanism where you create a collection of objects as if they were a single object. More precisely, you create a bounding rectangle which encompasses the sub-objects, and you use the existing layout attributes to lay out the container object.

CRiSP provides a variety of mechanisms to achieve this, mostly as a result of historical evolution, so the different mechanisms will be described in the order they were implemented. This gives a better understanding of when or how to use these attributes.

Conceptually, when you create a dialog box, you are creating a *list*, rather than a *tree* of objects contained in the dialog box. This list corresponds in order to the order you create your objects in the create_object() statement.

As CRiSP evolved, a mechanism for creating a sub-list evolved, in which instead of having a primitive object type as the basis of the nodes in the list, a T-bar junction was created. This T-bar effectively gives rise to a tree. The T-bar is known as a **group** object as it acts as a way of grouping a collection of sub-objects, for the purposes of the layout semantics.

For example, suppose we want a layout like this:

```
-----      -----
| A |      | C |
-----      |   |
-----      |   |
| B |      -----
-----
```

(where A, B and C represent three different objects, without different sizes). We normally would have to layout things left to right, top to bottom. With these rules we cannot possibly layout the above set of objects.

By using the grouping, what we can do is treat objects A and B as a single object, and now what we want to achieve is a layout of the two *objects* AB and C. To do this, we need to group object A and object B together.

This is achieved by using a pair of *brackets*, DBOX_GROUP_START and DBOX_GROUP_END. DBOX_GROUP_STARTs a level of grouping, and all objects declared up to the matching DBOX_GROUP_END are treated as part of the group. Groups can be nested, so arbitrarily complex layouts can be achieved this way. The above diagram might be implemented by doing something like:

```
obj_id = create_object(make_list(
        ...
        DBOX_GROUP_START,
                DBOX_BUTTON, "Object A",
                DBOX_BUTTON, "Object B",
        DBOX_GROUP_END,


        DBOX_LIST, quote_list("object C"),
                DBOX_NEXT_COLUMN, TRUE,
        ...));
```

The DBOX_GROUP_START declaration actually declares a new object, much the same way as DBOX_BUTTON or DBOX_LIST does. (The object created is called a DBOX_CHILD object). The syntax of the DBOX_GROUP_START and DBOX_GROUP_END words is pretty much like a set of parenthesis, so you can nest them.

The DBOX_CHILD object created as a result of a nested grouping can have attributes associated with it in much the same way as any other object. For example, you can center a grouping or place a grouping in the next space in the row.

Sub-groups

The DBOX_GROUP mechanism is explicitly designed to make it possibly to achieve certain types of layouts. By specifying DBOX_GROUP_START, you are effectively creating a new object, called a DBOX_CHILD. It is possible to create a branch off of the current object, without creating a place-holder object (DBOX_CHILD), by use of the attribute DBOX_SUB_GROUP_START ... DBOX_GROUP_END. A sub-group is very similar to a normal group, but was designed to solve a different problem, and first came to life with the implementation of the DBOX_STATUS_BAR attribute. As described previously, the menu bar, tool bar, and status bars are all similar in that they are actually a collection of appropriate sub-objects.

The grouping mechanism is used for the status bars, tool bars, and menu bars as a way of associating the individual components with the parent object.

Groups and sub-groups are very similar: the main difference is that a DBOX_GROUP_START declaration terminates the definition of the prior object and creates a new DBOX_CHILD, with all subsequent object definitions being a branch of the tree of the DBOX_CHILD (up until the matching DBOX_GROUP_END). A DBOX_SUB_GROUP_START declaration terminates the current object definition, but creates a new branch of objects, attached to the just created object (i.e. no DBOX_CHILD object is created).

The primary purpose of the sub-group is to support nesting of objects for the object types which need multiple components to create one user interface object.

Groups and the DBOX_CONTAINER object

The previous two sections have described the DBOX_GROUP_START and DBOX_SUB_GROUP_START mechanisms, and why they are needed. The third evolution of this functionality is the DBOX_CONTAINER object. As mentioned for DBOX_GROUP_START, when a new group is started, a new object is created, a DBOX_CHILD object. A DBOX_CHILD object has no semantics of its own, and in fact no widget or window exists for such an object. This is fine for the normal layout rules.

However, say you want to create a grouping of objects but you want to make them appear or disappear as and when needed? (The X11 API refers to this as managing or unmanaging a window; Windows refers to this is showing or hiding a window). You can create some interesting effects by creating a group of objects and not actually showing them; eventually when the user performs some action you *manage* or *show* the sub-objects all at once. A good example (and the first implementation of this) is the **Options→Language editing modes** menu. This dialog box consists of 5 sub-dialogs. The main dialog box has a fixed set of options at the left and the normal OK/Help/CANCEL at the bottom of the window. In the middle of the dialog box is a set of objects. The objects displayed is dependent on the set of options being looked at (e.g. File conversion, Language modes, Filters, etc). This effectively implements a tabbed-dialog box style of interface, although the look of the dialog does not resemble a tabbed dialog box.

The point of the DBOX_CONTAINER object is that all five layouts of this dialog box are created on invocation but only one of the layouts is made visible at any one time. (That is, you will see five DBOX_CONTAINER definitions if you examine the macro source in `src/crunch/gui/setup.cr`**).**

By creating sub-objects as children of a DBOX_CONTAINER it is very easy to manage or unmanage the container object itself rather than the sub-objects contained within the DBOX_CONTAINER object. The effect of this is that there is a near instantaneous change in visibility of the sub-objects. This same mechanism could be implemented without using a DBOX_CONTAINER but the code becomes not only more cumbersome to maintain (you need separate code to manage and unmanage each of the sub-objects), but also you will see a flickering effect (dialog box *in-fighting*) as the objects fight for real-estate before being pulled under the water.

The way this all works is to create a DBOX_CONTAINER object, just like any other object, but to use the DBOX_SUB_GROUP_START attribute to create a sub-grouping of objects which are all children of the owning container.

Properties (private symbols)

When you write a macro to create and handle a dialog box, one problem you will need to decide about is whether to allow multiple instances of that dialog box. Many dialog boxes in CRiSP are single instance dialog boxes, e.g. when you select a menu or icon entry to invoke a dialog box it will create that dialog box. Subsequent attempts to invoke that entry whilst the dialog box is on display will simply result in the existing dialog box being popped to the top of the display.

There is nothing preventing you from creating multiple instances of a dialog box, but you can end up needing a bit of state information to manage them. For a simple dialog box, you may be able to implement the functionality without the use of any global variables. Sometimes you will need to use gloabl variables to track the state of user selections, etc. But global variables are a real nuisance when it comes to dialog boxes that can be created multiple times - the values of the global variables becomes difficult to track. You can use a list structure instead of a primitive but this can be difficult.

Consider a very simple dialog box which has a single button and a label. The label displays a number and every time you click that button, the number is incremented. Where do you store the current value of the label? (You can use the label itself, but this is a special case, and doing so fails to illustrate the point being made here).

You can use a global variable, e.g. something like this:

```
int counter;


void
create_counter_dialog()
{
        counter = 0;
        create_object(make_list(
                DBOX_TITLE, "Counter Example",
                DBOX_CALLBACK, "counter_callback",


                DBOX_LABEL, "Current value: " + counter,
                        DBOX_NAME, "counter",
                DBOX_BUTTON, " Increment! "
                ));
}
void
counter_callback(int reason, int obj_id, int sub_obj, string name_id)
{
        change_object(obj_id, "counter",
                DBOX_VALUE, "Current value: " + ++counter);
}
```

This is fine but if you now create two dialog boxes at once then this will not work, since the two dialog boxes do not have their own *counter*.

As mentioned above, you could get around this problem by having a list of counter values but it is tedious to maintain the list, especially as the dialog boxes can be created and destroyed in an arbitrary order.

The solution to this problem is the use of a *property*. Analogues exist in other windowing systems, e.g. Windows allows the use of atoms, as does X11. A *property* is simply an arbitrary value associated with a specific dialog box. This is a symbol table containing arbitrary symbols (of any supported datatype). This symbol table allows you to stash dialog specific state information away and then later access and/or change the property values in your callbacks. This gets rid of the need for any global variables in your macro implementation and allows multiple instances of a dialog box to be created with very few problems.

Properties are created or set, via the *set_property()* primitive; they are retrieved with the *get_property()* primitive:

```
declare get_property(int obj_id, int rsvd, string name);


int set_property(int obj_id, int rsvd, string name, declare value);
```

Dialog boxes and Callbacks

The Microsoft Windows API and the Xt/Motif API are similar in a number of very basic areas, but the actually look and feel of the code is different. In a pure Windows program (without any MFC usage), you tend to see huge functions which handle the callbacks for specific dialog boxes, and these huge functions are typified by a large *switch* statement handling each of the possible actions the program implements. By contrast, Xt/Motif applications tend to be typified by lots of little callback functions, each callback handling the semantic actions of a single object in a dialog box.

Each approach has its advantages and disadvantages. (Having a single callback makes it easy to navigate code since you have a small number of callback functions to consider, and it is usually easy to pin point the one of interest; on the other hand, huge functions tend to blow compilers and optimisers and other source code analysers out of the water, and are not necessarily easy to maintain unless you follow some rigorous coding standards; With the Xt/Motif approach, you tend to have a lot of tiny functions, and it can be very difficult to find the function of interest, again unless you are very consistent with your naming conventions).

CRiSP takes a Microsoft Windows style approach. CRiSP is an interpreter and the macro language can best be considered as a very powerful language for configuring CRiSP. It is not necessarily designed to incorporate all the latest and greatest ideas in programming methodology, and hence takes poetic license to implement the most reasonable and easiest approach to the problems it wants to solve.

Macros in general are very high level programming statements. The mechanism CRiSP uses for supporting user actions on the dialog boxes is via a callback mechanism, whereby a single callback macro function is used to handle all input and events on a dialog box. This is achieved by using the DBOX_CALLBACK attribute, normally the second entry in the list of arguments to create_object() (immediately after the DBOX_TITLE attribute). The argument to the DBOX_CALLBACK attribute is a string which contains the name of a macro (and any optional parameters).

The macro is called with four parameters as arguments whenever an event occurs. The arguments are:

reason	This is an integer event code indicating why the callback function was invoked. (Similar to a Windows message or an X11 event type).
obj_id	This identifies the dialog box. This allows a single callback routine to handle an arbitrary number of dialog boxes.
sub_obj	This parameter is reserved, and is present for historical reasons. (Early versions of CRiSP referred to the objects in a dialog box by their ordinal position, but this lead to a maintenance problem as new objects were inserted into the dialog box).
name_id	This is a string corresponding to the DBOX_NAME of the object which triggered the event.

To handle the actions in the callback, it is simply a matter of deciding what the action was, and possibly which object in the dialog box caused the event to take place. (For example, you would need the name_id field if you had two or more push-buttons).

Platform Specific Issues

The CRiSP dialog box implementation hides most platform specific issues from the macro programmer. This means that the macros you write are highly portable with usually no changes required whether running under Microsoft Windows, X11/Motif or X11/XView. There are some things which may need to be borne in mind when writing dialog box macros - things which are available on some of the windowing systems, but not necessarily on the complete range.

One issue which is not immediately obvious is that of dismissing a dialog box. Most CRiSP dialog boxes contain a **CANCEL** button which is used to dismiss the dialog box without taking any further action. On each of the main windowing platforms, it is also possible to dismiss the dialog box by selecting the **CLOSE** option in the system (or window manager) menu. In this case the name_id parameter in the callback will be blank, but the callback reason will be **LCB_CANCEL**. You should always handle this message otherwise you may end up with undismissable dialog boxes which is usually very annoying (but quite common during dialog box programming and testing).

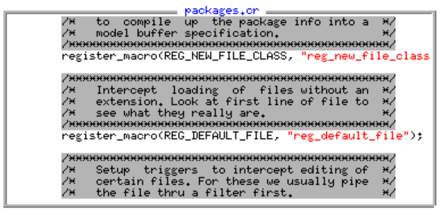
The size and shape of various objects is different on a per-platform basis. For example, Motif has very square buttons, whereas Microsoft Windows rounded buttons The fundamental idea of the dialog box system is to allow you to write dialog boxes without concern about the final appearance, yet at the same time knowing your dialog boxes will work on each system unchanged. Under some circumstances you may find yourself relying on the default layout semantics, but end up with horrible looking layouts in your dialog box, due to the difference in default sizes of the user interface objects.

How to create your own Colorization file

`Colorization`This section describes the CRiSP colorization facility, gives you information on how to create your own configuration and gives a couple of case studies to illustrate the points discussed.

The colorization process allows CRiSP to dynamically color certain parts of files, whether they be program source files, scripts or data files. Most files are actually very similar from a syntactical point of view: files contain comments, keywords, and string or number constants.

The beauty of colorization is that instead of looking at flat monochrome text, certain parts of your file are colored, which makes it easy on the eye to divide long files up into sectional components. For example, look at the following screen snapshot:



```
/*   to  compile  up  the package info into a  */
/*   model buffer specification.               */
/**********************************************/
register_macro(REG_NEW_FILE_CLASS, "reg_new_file_class

/**********************************************/
/*   Intercept  loading  of  files without an  */
/*   extension. Look at first line of file to  */
/*   see what they really are.                 */
/**********************************************/
register_macro(REG_DEFAULT_FILE, "reg_default_file");

/**********************************************/
/*   Setup  triggers  to intercept editing of  */
/*   certain files. For these we usually pipe  */
/*   the file thru a filter first.             */
```

This is a piece of CRiSP macro code. The CRiSP macro code looks and feels a lot like C. Without necessarily knowing any of the details of the macro language, you can immediately see that different parts of the edit window are colored differently. In the sample, comments have a grey background with black text. These comments act as eye-catchers, and nicely sub-divide long code sequences into regions of interest. It makes it much more each for the brain to look for patterns in color, rather than looking at the plain boring text.

It is important to realise that colorization is an **inexact** process. It relies on context independent information to guess the type of tokens in your file and color them according to your personal tastes. Having said that colorization is inexact, it is also important to note that colorization probably works in more than 99% of typical cases, so the results are actually very satisfying. The reason for highlighting this inexactness, is that if you decide to create custom keyword files, then you cannot expect to handle every situation which your files require. There are certain trade-offs involved. These trade-offs are to do with CRiSP's capabilities versus the ease with which you can create a custom keyword file.

There are two ways to create a colorization file. Using the Keyword Builder(pg. 68). dialog box on the Options menu, or by hand crafting a **.kwd** file. The Keyword Builder dialog provides most of the functionality required to create a custom colorization and is the preferred route. The other sections in this chapter of the manual provide a complete definition of the facilities available to create a colorization.

It typically takes 5 to 10 minutes to create your own file, which maybe can handle 90% of your colorization task. This is achievable usually by taking one of the provided keyword files, and customizing it for your own data file. To get a higher than 90% acceptable colorization may involve reading these sections more thoroughly to understand what all the options and features are.

CRiSP supports colorization by loading a keyword description file. These files usually have the file extension of **.KWD** and the ones CRiSP is supplied with are usually located in **/usr/local/crisp/src/keywords** or **C:\CRISP\SRC\KEYWORDS** for a Windows style installation. These files cover many languages, including Ada, C, Fortran, Perl, PostScript and Shell script. The differences between one language and any other is usually very small. If you look at the samples provided the main thing which you will see is the list of keywords. Other differences result in whether, for instance a language can support numbers in hexadecimal, or floating point, whether keywords are case sensitive, etc.

Each keyword file contains a description for a certain class of files. You can override any CRiSP keyword file, by creating your own version of the file and placing it in the personal configuration directory (**$CRCONFIG** or **$HOME/.Crisp**).

→ Keyword Builder dialog box(pg. 68).

→ Description of the .kwd keyword file format(pg. 70).

→ The flags= directive(pg. 71).

→ Character class directives(pg. 72).

→ Keyword directives(pg. 74).

→ Keyword flags(pg. 75).

→ Keyword regular expressions(pg. 76).

→ Limitations of Colorization(pg. 76).

→ Case study #1: The C Colorizer(pg. 77).

→ Case study #2: The Fortran Colorizer(pg. 78).

Keyword Builder

The keyword builder is a dialog which allows you to create and modify colorization definitions. A colorizer definition is needed to tell CRiSP how to perform colorization of keywords for files of a particular type. CRiSP comes with a variety of supported languages, but you can use this dialog to create your own or to modify a supplied definition. For example, you might be programming in C and want to add custom function libraries so that they are colored in a different color.

In order to create a colorizer there are three types of things you need to define:

Special parsing characters

Global attributes

Keywords

The list of *keywords* to define is relatively obvious. What is not obvious are the other two categories. In general you only need to modify the other two as you detect that things are not 100% accurate.

When you pop up the keyword builder dialog it will show you the current colorizer definition for the file you are viewiing.

Character classes

The character classes are used to define what a keyword looks like. For instance, in the C language, the acceptable characters making up a symbol are the upper and lower case letters, the digits (0-9) and the underscore. **ABC_DEF** is a perfectly valid symbol name. In some other language, maybe the underscore is not valid; in shell scripts, the dollar sign is used to introduce a symbol but can also terminate a symbol name, e.g. **$HOST$PWD**.

The character class definitions are pseudo-regular expressions containing character classes using the [..] notation. Refer to the Colorizer Character class(pg. 72). section for more details.

Global attributes

The global attributes describe aspects of the language you are about to colorize. For instance all keywords may be specified as case independent, or a special language syntax needs to be catered for. The attributes which you can specify are described in the flags(pg. 74). section.

Keywords

The term keyword is used to refer to the objects you want to be colored In your document. This includes string literals, comments, operators and normal keywords. You need to describe the keyword, give it a type (e.g. a comment) and specify any particular keyword specific attributes (such as comment ends at end of line). For plain textual keywords, things are easy - you just list the keyword, e.g. **default,** or **switch.**

Some keywords require more complexity in describing them; for example string literals in a program file cannot be described by simply listing the quote character. You need to tell CRiSP that a string literal is a sequence of characters starting with an open quote and terminating in the next non-backslashed quote.

The attributes which describe a keyword are similar to the global attributes, but sometimes have a slightly different interpretation. See the keyword flags(pg. 75). section for details on each of the available flags.

Creating a new colorizer

To create a new colorizer, follow these steps:

Invoke the Keyword Builder dialog.

Select the **New** button to create a blank dialog

Enter your language name in the **Language** input field.

Type in your keywords, setting the appropriate **Type** and flags for the keyword. Press **Add** to add the keyword to the keyword list. To modify a word already displayed in the table, double click on it to display the keyword. If necessary you can modify the word and add it in again or **Remove** it.

If you need to change the character class information or global attributes then select the appropriate tab of the dialog box.

When you have finished, select the **Save** button to save the colorizer associated with the previously specified language name.

Dialog box buttons

New	Clears the dialog in preparation for typing in a new definition. You are prompted to save any changes before the dialog box is cleared out.
Save	Saves the current definition and closes the dialog box.
Add	Appends the current **keyword** along with the flags and **type** fields to the keyword table.
Remove	Deletes the currently selected word in the table.

.KWD Keyword File Format

`Colorization:.kwd file format`CRiSP stores colorizer or keyword descriptions in a special file, called a .KWD file. Keyword files have a **.kwd** file extension. You can see all the CRiSP supplied colorizer descriptions in the installation directory under the **src/keywords** directory.

A keyword file is a plain text file, with the following syntax:

→ Comments are bracketed between **/*** and ***/**. I.e. they are similar to C comments.

→ A keyword file can contain multiple keyword sections. Each section defines the colorizations for a single file type. Each section starts off with the file type in square brackets.

→ Following the section name is a list of directives defining various attributes of the grammar. The various keywords which make up this section can be provided in any order.

The following example is a cut down version of the C colorizer file illustrating many of the features of the colorization process:

```
[c]
flags=c_hexadecimal c_floats

char_start="[A-Za-z_]"
char_next="[A-Za-z_0-9]"
char_new_token="[/*]"
char_operator="[/*]"

directive="^[ \t]*#*"

comment=accept_backslashes "//*.*$"
comment="/**.**/"

string=accept_backslashes "\"*.*\""
string=accept_backslashes "'*.*'"

/********************************/
/*      C keywords.             */
/********************************/
keyword="asm" "auto" "break" "case"
keyword="char" "const" "continue"
```

Comments

Keyword files can contain comments to help annotate what you are doing, or to put copyright or revision history in. Comments are like C comments, they start with **/*** and everything up to the next ***/** constitutes the comment. They do not nest.

Section Name

The section name appears on a line on its own inside square brackets. Generally speaking this name would be the same as the basename of the filename without the file extension. E.g. C keywords are stored in **c.kwd** and the section name is **[c]**.

The section name and filename do not need to agree, but it is best to pick some scheme that won't lead to confusion. For example, CRiSP macro files generally have a **.cr** file extension, but the keyword filename is **crisp.kwd**, and the section name is **[crisp].** In the language setup menu, you are presented a list of colorizers available, based on the **filename** not the section name.

The flags= Directive

`Colorization:flags directive`The **flags=** entry in a keyword section are used to provide various hints about the language being colorized. Each hint is a word; see the list below for a description of the various attributes defined. There can be multiple **flags=** lines. Use whatever layout best suits you.

The various keywords described for the **flags=** directive may also be specified with the actual keyword definitions as well. Not all combinations of flags are meaningful. For example, you can specify that string constants allow backslashes inside them (**accept_backslashes).** See the sections on handling the **keyword=, string=,** etc directives.

accept_backslashes
: This says that the backslash (****) character is used within the language to turn off the special meaning of certain characters. For example, in a Shell script, the backslash character would be used to specify a single quote as in:

  echo \"

  Without this attribute, the backslash would be ignored and the double quote character would indicate that a string was starting.

accept_double_char
: Use this in a language which uses a doubling-up of quote marks to indicate a single quote in a string. For example, whereas in C, to define a single apostrophe char constant would be specified as: **'\' '**, in Ingres Windows 4GL, you would specify **' ' ' '**, i.e. four consecutive apostrophes.

c_floats
: This specifies the language supports floating point numbers, in the same format as the C language. This allows for an exponent and a fractional part. The C floating point convention is suitable for nearly every programming language, including Ada and Fortran. Although Fortran accepts a double precision exponent (e.g. `1.234d+04`), CRiSP does not support this.

c_hexadecimal
: Use this flag if the language supports the specification of hexadecimal numbers in a manner similar to the C language, e.g. a leading 0x followed by the hex digits.

case_independent
: Use this flag if the language accepts keywords without regard to case. C is a case-sensitive language, whereas Fortran is not. SQL is another language which is not case sensitive.

eol_terminate
: This flag is used to mark string constants which should be terminated if the end of line is reached before a matching close delimiter is found.

  Without this flag, the space between the end of the line and the start of the next line would be colored in the string color. With this flag, the coloring is turned off at the end of the line defining the start of the string constant.

  The **perl** and **AML** colorizers use this facility because of ambiguities in parsing their source files.

embedded
: This flag is not usually specified as part of the **flags=** definition, but is used in keyword definitions to indicate that a keyword may start inside a word.

no_numbers
: Set this hint if your language has no notion of numbers. This is used for example in the nroff/troff colorizer; this avoids numbers being colored as numbers. Instead they are treated as part of the normal text.

spell
: Specifies that the default setting for all keywords is for spell checking. You would not normally use this as it would cause keywords (which are usually not valid words in your native language) to be underlined showing a mis-spelling. But you might use the **spell_text** keyword below.

spell_text

For most languages, you would not spell check all the types of entities in your files. You would normally restrict spell checking to things such as comments or quoted literal strings. For those types of languages you explicitly state (using the **spell** keyword) that each syntactic item needs spell checking.

In the case of a language like HTML, you would want to check the raw text but not check any specific keyword definition; this is almost the direct opposite of other programming language files. Instead of checking (or as well as) the spelling for comments and keywords, you also want to spell check the actual text which isn't defined as a specific keyword type. To achieve this, you need to specify the **spell_text** keyword in the flags definition. This indicates to CRiSP that the default action for non-keywords is to spell check.

The HTML colorizer definition (**src/keywords/html.kwd)** includes this setting as standard.

start_of_line

This flag has no meaning for the **flags=** directive. It can be used with keyword definitions to indicate that they are only valid at the start of a line.

x_hexadecimal

Specify this flag if hexadecimal numbers are represented using the notation: **X'nnn'** where **nnn** is a hexadecimal string.

Character Class Directives

`Colorization:character classes`CRiSP has a minimal idea of what constitutes a keyword. If you look at normal plain text, the human brain can easily decipher what constitutes a word, e.g. a sequence of letters. White space and other punctuation delimits a word.

Consider a language like C. If we applied the same definition then we would end up treating a variable name like:

```
pointer_to_list_array_struct
```

as 5 individual words, with the word **struct** at the end of the line as a keyword and not a variable.

Different languages have different naming conventions for keywords and variable names. Many are similar to the C definition, maybe with some slight alteration. In the C language, a variable is a sequence of upper and lower case letters, digits or the underscore character, with the restriction that variables cannot start with a digit. (If variables could start with a digit, then we would have problems distinguish a variable from a number when only looking at the first character).

The keyword description file allows you to define character classes, i.e. sets of characters, which form part of the syntax of the language.

All of these character class directives start with the prefix **char_**. A character class definition allows you to specify the valid characters within certain contexts of the language. CRiSP uses these contexts to resolve ambiguities it comes across.

Character classes are specified as a regular expression style character class, by enclosing the list of characters in square brackets. The entire string is specified in string quotes. For example

```
char_start="[A-Za-z_]"
char_next="[A-Za-z_0-9]"
char_new_token="[/*]"
char_operator="[/*]"
```

Most languages will use a simple variation on the theme above (the above is for the C language).

char_start

This defines the characters which are start a keyword or variable name. (As far as CRiSP is concerned, the only difference between a keyword and a variable name is the color it is drawn in).

For example in C, this is the letters and underscore: **[A-Za-z_].**

char_next

Defines the characters which can form the 2nd and subsequent characters in a keyword or variable name. This may be different from the **char_start** field.

For example in C, this is the same as the first character, but also includes the digits: **[A-Za-z0-9_]**.

char_last

Specify this to indicate characters which are a valid part of a keyword or variable name. Whereas **char_start** and **char_next** specify characters which form part of the name, **char_last** also specifies that any of these characters terminate the name.

For example, consider a Bourne shell script. In the script language, you can define variables, and they are usually specified with a preceding dollar **$** character. The syntax for Bourne shell is peculiar in that you can create arbitrary variables using this syntax, but there are certain variables built into the language. For example, **$$** is the current process id. In this case, a subsequent **$** is to be treated as part of the variable name, but also to terminate the name at that point.

An example might be:

```
char_start="[A-Za-z_$]"
char_next="[A-Za-z0-9$]"
char_last="[$]"
```

char_new_token

This character class is designed for languages like shell script where there is an ambiguity in the end of one keyword and the start of the next.

Consider a statement like:

```
echo $DIR$FILENAME
```

In this example, after the keyword **echo** we have two separate variables, **$DIR** and **$FILENAME**. If you look at the **char_last** description given above, then we see that the above could be parsed as **$DIR$** followed by **FILENAME**. If we had something like **$DIR$echo** then the word **echo** would be treated as a keyword, and not as a variable.

The **char_new_token** class tells CRiSP that characters in this class are part of a token, but also start a new token, even if we are already inside a token. Thus, a more correct definition for a shell script colorizer would be:

```
char_start="[A-Za-z_$]
char_next="[A-Za-z0-9$]
char_last="[$]"
char_new_token="[$]"
```

char_operator=

Use this character class to resolve ambiguities concerning characters which have dual roles in the language. This is mainly used in C like languages where there is a conflict on the forward slash character **/**. A forward slash in C can be used in three or four parts of the language:

l If followed by an asterisk, then we are at the start of a multi-line comment. If followed by another slash, (assuming C++ comments), then it is a comment until the end of the line. Otherwise it could be a divide or divide-eq operator.

CRiSP supports the ability to color operators (generally an operator is something that isn't a keyword, a number, a comment or a string) in a separate color. Because the slash character is part of a comment, CRiSP would normally believe a slash to be a valid symbol character. By specifying a character class for operators helps to avoid mis-coloring a slash which is not the start of a comment.

Consider the following example:

```
int x = 3 /sizeof y;
```

Without the **char_operator** class definition, because there is no space between the forward slash and the keyword **sizeof**, then CRiSP would assume that we had a variable called **/sizeof** and not color the keyword as you would expect.

The following is an example of the **char_operator** directive for the C language:

```
char_operator="[/*]"
```

char_backslash=

This class is used to redefined the backslash character. In many languages, backslash character is used to turn off (or *escape*) the meaning of the following character. For example in a shell script or the C language, this mechanism allows you to embed a quote inside a string, which would otherwise normally terminate the string.

Whether your language will use backslashes or not is handled by specifying the **accept_backslashes** keyword prefix, e.g. so that backslashes are handled in a context sensitive manner.

You should use this sequence only if you want to use an alternate character other than the backslash to have this meaning. If you do use this character class, then the backslash will lose its normal meaning, and if you want to maintain the backslash then ensure you add it to the character class.

Keyword Directives

`Colorization:keyword directives`CRiSP supports for coloring comments, strings, numbers, reserved words and other artefacts of a file. The general term used to describe these entities is a **keyword**. When looked at closely, the only difference between these token classes is simply the default color in which the tokens are displayed.

Because all these user definable types are the same, the mechanism to describe them is the same; the only difference is the directive used in the **.kwd** file to define the class of the keyword, and thus affect the coloring.

The following classes of keywords are available:

→ **string** used for things that looks like string constants or literals.

→ **comment** used for comments. Comments can be multi-line or extend to the end of a line.

→ **keyword** used for keywords and reserved words in the language.

→ **directive** a special context sensitive type of keyword construct, e.g. the directives used in the C preprocessor are a good example.

→ **symbol1, symbol2, symbol3,** and **symbol4**. These are *spare* classes which you can use for other categories of syntax elements, where the existing categories are insufficient. For example, in SQL, **symbol1** and **symbol2** are used to color global and local variables. (This is doable in SQL, because the syntax of a variable allows you to determine its scope; it is not possible in C, for example, because a semantic analysis of the file would be necessary to determine the storage class of a variable).

The format of a directive line is exemplified by the following:

```
directive="^[ \t]*#*"

comment=accept_backslashes "//*.*$"
comment="/**.**/"

string=accept_backslashes "\"*.*\""
string=accept_backslashes "'*.*'"

/*******************************/
/*      C keywords.            */
/*******************************/
keyword="asm" "auto" "break" "case"
keyword="char" "const" "continue"
```

There are a number of things to note about these definitions:

→ A directive starts on a line with one of the keywords **directive=, comment=, string=, keyword=, symbol1=, symbol2=, symbol3=** or **symbol4=**.

→ More than one keyword can be specified on each line; each keyword is enclosed in double quotes. Backslashes are used inside the quotes to escape the next character. (Usually only needed for literal double quotes or backslashes.

→ Keywords are case dependent **unless** the **case_independent** flag is applied to the keyword or specified in the **flags=** line.

→ Each keyword can be preceded by one or more flags. The keyword specific flags are described below. The meaning of these flags only applies to the next keyword, and is turned off for subsequent keywords.

The text for a keyword is taken literally, but certain regular expression characters can be used to give more powerful meaning to the construct. These regular expression characters are a limited subset of the complete form of regular expressions that are available when performing search and replacing operations.

Keyword Flags

Colorization:keyword flagsThe following flags can be used when specifying keywords to affect the normal meaning of the keyword:

accept_backslashes
Normally used for things like string constants. When this is used, it is usually used for a keyword which is defined as a pair of strings, e.g. a start string character, and an end string character. With this flag in effect it means that a backslash inside the keyword turns off the effect of a special character. For example in the C definition of a string constant, a backslash can be used to include a double quote in the string constant itself, and hence does not terminate the string.

accept_double_char
Use this for string constants or any other paired string, where two occurrences of the starting character are to be treated as part of the keyword and not a terminator. (This only makes sense with comments or string constants where the starting and ending characters are the same).

case_independent
You can specify that some keywords are case sensitive or not by using this keyword. You can specify that all keywords are case independent by using this flag in the **flags=** section.

embedded
This flag means that the keyword can be embedded inside any arbitrary text. For example, say you have a file containing a DNA sequence. This file may consist of a sequence of letters like this:

```
GGTTTGAACATCATCATCATCGGGAAAAATTTT....
```

You might want all the sequences **ATC** to be treated as keywords and hence colored differently from the rest of the text. You wouldn't normally be able to do this because CRiSP would treat the whole string as a single keyword. By specifying something like:

```
keyword=embedded "ATC"
```

means that as soon as the sequence **ATC** is seen, then it will be colored even although it is sitting inside a higher level keyword.

regexp
If this keyword is present, then the keyword string will be treated as a Unix style regular expression rather than the limited style regular expression described elsewhere. Using regular expressions is a slower way of specifying a keyword, but is useful to handle some of those exceptional syntax cases in a language. For example, in the **CHILL** language, a hexadecimal constant can be described as:

```
keyword= regexp "H'[0-9A-F]*"
```

spell Use this attribute to enable spell checking. You wouldn't normally use this for an explicit keyword, but would use for keywords defined as expressions, e.g. comments, string literals.spell:colorizer flag

---

Keyword Regular Expressions

`Colorization:keyword regular expressions`Keyword regular expressions are a way of allowing generic styles of strings to be specified as keywords or string constants, etc. In a language such as C, it is obvious that something like **"case"** specifies a keyword, but it is more difficult to describe a string literal because although the start and end of the literal are well defined, anything can occur inside the quote marks delimiting the string.

CRiSP allows limited regular expressions to be used to define this vagueness in a keywords definition. These are not full-featured expressions as you can use in the normal search and replace.

The following describes the character sequences which can be used in a keyword.

^ If this character is used at the start of a keyword, then that keyword will only be recognised at the start of a line. For example, Fortran comments (when using the **C** notation are restricted to starting at the start of a line.

***** The asterisk operator has special meaning when following the start of a keyword. It is used to indicate that anything can follow as the next character. It is used to avoid ambiguities. For example, consider the Fortran **C** comment again. A Fortran comment using this notation starts at the beginning of a line and because it is a comment, then anything can follow, even another **C.** CRiSP would normally treat the string **C** and **CC** as two distinct keywords, so the normal rules wouldn't work.

By specifying:

```
comment="^C*"
```

you would achieve the desired effect. However, this example is not totally correct. (See below).

.* This sequence means 'any string of characters' it used when defining string constants or comments, where you specify the starting character(s) and terminating character(s), but allow any arbitrary text between these characters.

You can only use one occurrence of this expression in a keyword definition. You cannot define a context sensitive keyword such as: **ABC.*DEF.*GHI** as that would require looking ahead to validate the keyword.

$ This expression can be used at the end of a keyword definition. It is normally used in conjunction with indefinite keywords, such as comments or string literals which can extend indefinitely. For example the C++ ***//*** comment extends to the end of the line, so the definition is:

```
comment="//*.*$"
```

[\t]* or [\t]*

These two sequences are special. Although they look and feel like character class wild cards, they have a very limited scope. Either one of the two forms may be used after a caret (**^**). It was added to support C preprocessor directives which normally start at the beginning of a line. These preprocessor directives can be preceded by an arbitrary amount of white space, so the leading white space is taken into account when colorizing.

These sequences will only work in this restricted area.

Limitations of Colorization

`Colorization:limitations`Colorization first appeared in CRiSP version 3. It has undergone considerable changes in each subsequent release. The things which have consistently changed over the releases is that more and more expressive power has been added to handle the vague corners of various different languages.

For example, in the C programming language, a string constant consists of an opening quote (single or double depending on whether it is a string constant or a character constant), and is followed by the text of the string, with a matching single or double quote to terminate the string. If you want to include a quote inside the string, or any other non-printable character, you can use a backslash to affect the following characters.

In many languages, string constants are handled differently. In ADA, strings are enclosed in single quotes, and to quote a single quote, you just specify three apostrophes in a row.

There are other coding differences from one language to another. CRiSP has evolved to cater for many of these different styles, but it is not generic enough to allow any kind of quoting policy.

Colorization is a process which analyses a file based on a **syntax** specification. This means that CRiSP is looking at very little context to determine the meaning of any part of the text. For example, if CRiSP sees a digit, it reasonably expects the digit to be part of a number, with successive digits and possibly a fraction to follow (unless told otherwise)

CRiSP does **not** perform a **semantic** analysis of a file. Semantic analysis means parsing an entire file, and determining that the exact order of words, keywords, constants etc., are valid or not. Take the following example:

```
int fred = 3;
```

This is a statement in the C or C++ languages which creates an integer variable called **fred** initialised to contain the value **3.** From a syntax point of view, there are 5 tokens: **int,** a symbol**,** an **=** operator, a number 3, and a semicolon.

From a semantic point of view, the above statement is correct. Now consider this re-ordering

```
fred = ; int 3
```

This is total nonsense to a C or C++ compiler. CRiSP can colorize this nonsense quite happily because it doesn't worry about the order and context of the tokens. It simply identifies each token by the first character or so.

Thus you cannot specify a language in terms of valid statements, only valid keywords.

The reason for not performing a semantic analysis of your code is that this would be too slow, and many software development tools excel in doing this for you.

The reason for emphasising this point is that for non-programmers, it may be difficult to understand what is possible and what is not possible.

Case study #1: C colorizer

Colorization:case study 1The following example illustrates the C colorizer provided with CRiSP. The original file can be found in the distribution directory **src/keywords/c.kwd.**This colorizer file demonstrates most of the features and facilities of writing a colorizer.

```
[c]
flags=c_hexadecimal c_floats

char_start="[A-Za-z_]"
char_next="[A-Za-z_0-9]"
char_new_token="[/*]"
char_operator="[-^!|&<>~()/*+=?:]"

directive="^[ \t]*#*"

comment=spell accept_backslashes "//*.*$"
comment=spell "/**.**/"

string=spell accept_backslashes "\"*.*\""
string=accept_backslashes "'*.*'"
```

```
operator= "+=" "-=" "*=" "/=" "|=" "&=" "^=" "==" "!=" "<<=" ">>="
operator= "+" "-" "*" "/" "^" "&" "|" "!" "~"
operator= "?" ":" "++" "--" "(" ")" "[" "]"
operator= ">" ">=" "<" "<=" "->" ">>" "<<="

/*****************************************************************/
/*   C keywords.                                                 */
/*****************************************************************/
keyword="asm" "auto" "break" "case" "char" "const" "continue"
keyword="default" "do" "double" "else" "enum" "extern"
keyword="float" "for" "goto" "if" "int" "long" "register" "return"
keyword="short" "signed" "sizeof" "static" "struct" "switch"
keyword="typedef" "union" "unsigned" "void" "volatile"
keyword="while"
```

Case study #2: Fortran colorizer

`Colorization:case study 2`The following example illustrates the Fortran colorizer provided with CRiSP. The original file can be found in the distribution directory **src/keywords/fortran.kwd.** This colorizer file demonstrates most of the features and facilities of writing a colorizer.

```
[fortran]
flags=case_independent c_floats

char_start="[A-Za-z_]"
char_next="[A-Za-z_0-9$]"

string="\"*.*\""
string="'*.*'"

comment=spell "!*.*$"
comment=spell "^**.*$"
comment=spell "^C*.*$"

keyword="backspace" "block"
keyword="call" "character" "close"
keyword="common" "complex" "continue"
keyword="data" "dimensin" "do"
keyword="else" "elseif" "end" "enddo" "endif" "endmap"
keyword="external" "equivalence"
keyword="format" "function"
keyword="goto"
keyword="if" "implicit" "include" "integer" "intrinsic"
keyword="logical"
keyword="map" "namelist"
keyword="open" "parameter" "precision"
keyword="print" "program"
keyword="read" "record" "real" "return" "rewind"
keyword="save" "stop" "structure" "subroutine"
keyword="then" "type"
```

```
    keyword="union"
    keyword="while" "write"
```

Interprocess Communication and CRiSP

Interprocess communication is the means to allow other programs to communicate with CRiSP, or vice versa. IPC communication is a very powerful mechanism for extending an existing application or CRiSP. For example, CRiSP provides IPC communication when dealing with shell buffers, compilation commands and filtering.

Probably the most frequently asked question by users is how to tell CRiSP to edit a file automatically and position itself to view a particular line in the file. This is easily achieved using any of the IPC mechanisms described below.

CRiSP provides a variety of IPC mechanisms because there is no one universal best IPC mechanism - each mechanism has its own advantages and disadvantages.

The following sections describe the IPC mechanisms in more detail and how to program them in the CRUNCH language.

→ IPC Mechanisms(pg. 79).

→ IPC Primitives(pg. 80).

→ IPC Callbacks(pg. 80).

→ TCP/UDP Communications(pg. 81).

→ Pipe Communications(pg. 84).

→ PTY Communications(pg. 84).

→ DDE Communications(pg. 85).

→ Signal Communications(pg. 86).

IPC Mechanisms

CRiSP supports one or more of the following IPC mechanisms. The actual mechanisms supported are platform specific.

TCP/UDP(pg. 81). TCP/UDP is probably the best communication mechanism. Using either of these protocols allows communication between CRiSP and applications either on the same machine or remote machines.

From a programming perspective, both protocols are very similar. The major difference is that TCP is a connection oriented protocol, meaning that you have to establish connection with another task on the network before communication can commence. The resulting connection is *reliable* in the sense that the underlying protocol architecture does its best to handle errors in the data stream and retransmits lost packets.

UDP is connectionless and unreliable. If a packet gets lost somewhere on the network, it is up to the applications using the protocol to discover and take remedial action. UDP is more lightweight than TCP and is more suitable for certain classes of communication, e.g. statistical status monitoring.

Pipes(pg. 84). Pipes are a communication mechanism that is suited for communication between two tasks on the same machine.

Under Unix, there are two types of pipes - anonymous pipes and named pipes. Anonymous pipes are suitable for a parent process to talk to a child process. Named pipes are very similar but are more suitable when there is no parent-child relationship.

Windows/32 supports anonymous and named pipes as well. CRiSP does not currently

	support these pipes. Windows/32 named pipes are more like TCP connection which allow two tasks on different machines to communicate.
PTYs(pg. 84).	A PTY, or pseudo-teletype, is the mechanism used by terminal emulators under Unix and X-Windows, to allow the user to have separate xterm windows. Some programs can be communicated to via pty's more easily than using any other IPC mechanism. For example, the *vi* editor under Unix has no IPC support in it. But you could create a PTY running vi and send keystrokes to the task.
DDE(pg. 85).	DDE (Dynamic Data Exchange) is a mechanism only available on the Windows platform. DDE is a special mechanism for interprocess communication which is limited to communicating between tasks on the same machine in an unreliable manager. (Unreliable means that there is no guarantee that a DDE operation will be successful; e.g. running out of resources can cause an operation to fail). DDE has mostly been supplanted by much higher level protocols, such as OLE. However DDE is most commonly used in SETUP applications for Windows applications to talk to the Program Manager or Explorer to create program groups for installed applications. DDE can be used, for instance, also to communicate with the WEB browsers (such as Netscape's Navigator or Microsoft's Internet Explorer product), e.g. to tell it to fetch particular pages.
Signals(pg. 86).	Under the Unix operating system, CRiSP supports signal notification using the **SIGUSR1** and **SIGUSR2** signals. This allows a macro to be notified of some external event.

CRiSP provides access to these IPC mechanisms using a generic mechanism. Macros can be written to largely ignore the distinction between any of these IPC types.

{button See Also, ALink(ipc,,,)}

IPC Primitives

CRiSP provides the following primitives for accessing the IPC mechanism:

ipc_create	Used to create a connection.
ipc_close	Closes a connection.
ipc_read	Read data from a connection.
ipc_write	Write data to a connection.
ipc_accept	Accept an incoming connection request from a remote application.
ipc_status	Retrieve miscellaneous status about a connection.
ipc_register	Register a callback which can be called when data is available to be read, written, or on an exception.
ipc_unregister	Cancel a callback.
register_macro	A generic function used to register CRiSP callbacks on certain conditions. One of the important conditions in the area of IPC is the REG_SIGNAL condition.

These primitives are loosely similar to the underlying socket library available on most systems. The primitives provide sufficient generality to write non-blocking macro applications that can act transparently within the CRiSP editing environment.

These primitives are fully covered in the CRiSP Macro Primitives Guide.

{button See Also, ALink(ipc,,,)}

IPC Callbacks

The way to successfully use the IPC primitives in a macro is based on callback macros. You can do things synchronously in line, but doing so can cause the user interface to hang whilst waiting to send or receive remote data.

When using the TCP/UDP protocols, you may face long delays in making a connection to a remote host. All kinds of things can cause this delay - propagation delays, DNS name resolution, and machines which are down.

The key to writing non-blocking macros is to write them based on callbacks, using the ipc_register() function. There are four conditions for which you can register a callback:

IPC_TRIGGER_READ`IPC_TRIGGER_READ`

indicates data is a available to be read. You can successfully call ipc_read() **once** to read data. If you attempt to call ipc_read() more than once within a callback you may block if the no data is available.

If you back a zero length string then the connection has been disconnected by the remote end.

IPC_TRIGGER_WRITE`IPC_TRIGGER_WRITE`

indicates you can write data down a connection in a non-blocking manner. Normally you would not need to worry about this condition, but if you plan to send large amounts of data then you may need to take this into account to avoid causing CRiSP to hang on an ipc_write() function call.

When the callback is invoked you can call ipc_write(). You are advised to use the IPC_NON_BLOCKING flag on the call to ipc_create() to ensure that the ipc_write() does not block. If you are attempting to send too much data then the ipc_write() call will return a value less than the lentgh of the data you are sending.

IPC_TRIGGER_EXCEPTION`IPC_TRIGGER_EXCEPTION`

This trigger is not guaranteed to have meaningful semantics for all the IPC communications mechanisms, and is designed mainly for the TCP protocol.

IPC_PROCESS_DEATH`IPC_TRIGGER_PROCESS_DEATH`

This trigger is called for the IPC_ANON_PIPE and IPC_NAMED_PIPE IPC mechanisms. It is used to indicate that the child process has terminated. You can usually ignore this callback since you can detect the end of a communications session by having an IPC_TRIGGER_READ callback invoke and attempt to read from the IPC connection. A zero length return from ipc_read() most likely indicates the session has terminated. `IPC_ANON_PIPEIPC_NAMED_PIPE`

When using the registered macro callbacks, you should set up the callback as soon as possible after creating the IPC connection. If you attempt to set it up after the current macro has returned, then you may miss a callback condition. This will happen if your macro terminates and CRiSP goes back to it's internal main loop to read keyboard or mouse input.

{button See Also, ALink(ipc,,,)}

TCP/UDP Communications

When using TCP or the UDP protocol, you can create a client or a server connection end point. The most common type is a client end-point. In the client scenario, CRiSP will connect to some service on the network, for instance an HTTP WEB server. A server is the thing which provides a service, e.g. a printer server, WEB server, FTP server.`IPC_TCPIPC_UDPTCP/UDP programming`

You can create both types of connections. To create a TCP connection you use the ipc_create function. The format is:

```
int ipc_id = ipc_create(IPC_TCP, "host:port");
```

```
int ipc_id = ipc_create(IPC_UDP, "host:port");
```

The second parameter to the ipc_create() primitive is the name and port number of the service. The name field can be a standard hostname, e.g. **crisp.demon.co.uk**, or an IP address in standard notation (e.g. 192,80,255.255). The port number is a standard TCP or UDP port number or service name. E.g. the following are valid names:

crisp.demon.co.uk:ftp

crisp.demon.co.uk:23

192.8.99.99:23

If you are creating a server application then you should omit the hostname and separating colon. For example

```
int ipc_id = ipc_create(IPC_TCP, "ftp");
```

The above example would create a TCP connection service which listens on FTP port number.

Client Side Programming Issues

When using TCP to connect to a remote site, there may be some considerable delay in the connection being accepted, or the remote site may be down, in which case you can hang CRiSP until the connection request completes or the connection times out.

You can avoid this problem by OR-ing the IPC_NON_BLOCKING flag into the IPC_TCP connection type. In this case, the ipc_create() function will return immediately and you can register a notification callback to tell you when the connection has completed. If you use the IPC_NON_BLOCKING parameter, you should delay reading or writing data down the connection until you are sure the connection has been established properly.

Server Side Programming Issues

For a server, the issues are similar. A server can handle multiple incoming connections at the same time. To achieve this you can register a connection notification. When an incoming connection is received, you then issue the ipc_accept() function. This creates a brand new connection and leaves the original connection handle ready for subsequent connection attempts.

Examples

The following example code illustrates a client making a request and sending a single message - all in a blocking manner. The macro will not terminate until the (possibly long) connection has completed:

```
void
client_example1()
{
	int	ipc_id;

	if ((ipc_id = ipc_create(IPC_TCP, "remotehost:1234")) < 0) {
		message("Connection failed, errno=%d", errno);
		return;
		}

	ipc_write(ipc_id, "Hello\n");

	ipc_close(ipc_id);
}
```

The following example illustrates the same code as above, but without blocking on the ipc_create():`IPC_TRIGGER_WRITE`

```
void
client_example_non_block()
{	int	ipc_id;

	if ((ipc_id = ipc_create(IPC_TCP | IPC_NON_BLOCKING,
		"remotehost:1234")) < 0) {
		message("Connection failed, errno=%d", errno);
		return;
		}

	message("Awaiting connection completion...");
	ipc_register(ipc_id, IPC_TRIGGER_WRITE,
"connection_callback");
}

void
connection_callback(int ipc_id)
{
```

```
            /* We have either successfully or unsuccessfully */
            /* connected to the remote host. */
            /* We can tell the difference by whether the ipc_write */
            /* is successful or not. */
            if (ipc_write(ipc_id, "Hello\n") < 0) {
                  message("Writing data failed.");
                  }

            /* If we keep the connection open at this point, make */
            /* sure to unregister the IPC_TRIGGER_WRITE trigger */
            /* otherwise it will keep firing. */

            /* In this example, we simply close the connection when */
            /* we are done. */
            ipc_close(ipc_id);
      }
```

The following example illustrates creating a service which other applications can connect to.

```
      void
      server_example()
      {     int   ipc_id;

            if ((ipc_id = ipc_create(IPC_TCP, "1234")) < 0) {
                  message("Failed to create server port");
                  return;
                  }

            /* Register macro to handle new incoming connections. */
            /* Note: we pass in the ipc_id as an argument so the */
            /* callback knows which connection to use. */
            ipc_register(ipc_id, IPC_TRIGGER_READ,
                  "new_connection " + ipc_id);
      }
      void
      new_connection(int ipc_id)
      {     int   ipc_id2;
            stringstr;

            if ((ipc_id2 = ipc_accept(ipc_id)) < 0) {
                  message("Failed to handle new connection request.");
                  return;
                  }

            /* Read message request from the remote client */
            str = ipc_read(ipc_id2);
            message("Received '%s'", str);

            /* Close down the connection */
            ipc_close(ipc_id2);
      }
```

{button See Also, ALink(ipc,,,)}

Pipe Communications

There are two forms of pipes which can be used: anonymous pipes and named pipes. Both types are very similarly - the difference is in the naming conventions used by the client and server to connect to each other. Presently, these two forms of communication are only implemented under Unix.`PipesIPC_ANON_PIPEIPC_NAMED_PIPE`

In order to use pipes, you need to execute some other process with its stdin and stdout set to a pair of file descriptors which are used by CRiSP to communicate with.

The following example shows how to create an anonymous pipe connection to a process:

```
int    ipc_id = ipc_create(IPC_ANON_PIPE, "date");
```

The named pipe version of the same thing would be:

```
int    ipc_id;

mkfifo("/tmp/pipein");
mkfifo("/tmp/pipeout");
ipc_id = ipc_create(IPC_NAMED_PIPE,
            "/tmp/pipein",
            "/tmp/pipeout",
      "date");
```

If you are using named pipes then you will need to create the named pipe files in the file system. You can do this with the mkfifo() macro primitive.

Assuming the program to execute starts up correctly, then you can proceed to use the ipc_read() and ipc_write() functions for communicating with the task, and the ipc_register() function for handling callbacks.

You can detect the death of the child process by attempting to read from the IPC channel. If you read back zero bytes, then the other end of the pipe has most likely terminated. Alternatively you may use the ipc_register() callback to register notification on the death of a process.

{button See Also, ALink(ipc,,,)}

PTY Communications

PTY communication is similar to using pipes for IPC communication. PTYs are only implemented under Unix, and it is this mechanism which allows Unix to support remote logins and multiple command terminal windows under X-Windows. (By contrast, Microsoft Windows does not support the PTY concept which is why you cannot telnet or rlogin into a Windows based machine).`PTYPseudo terminal:IPCIPC_PTY`

Using a pty for communication is best suited to very specific types of applications - applications which use stdin and stdout for user interaction, but for which using a pipe may cause a problem.

One area where a PTY may be more useful than a pipe is for a program which does buffered output. Most programs use the *printf()* function to display messages to the user. The default ANSI C library sets all output from a program to the terminal to be line buffered, whereas output to a disk file or pipe is block buffered. This can cause problems when using some programs with a pipe as you may not see any output from the program until sometime after you have typed something in to them.

You can get around this problem by using a PTY which fools the program to believe that it is directly connected to a terminal and thus behave correctly.

The way to create a PTY connection is as follows:

```
int    ipc_id = ipc_create(IPC_PTY, "date");
```

Assuming the program to execute starts up correctly, then you can proceed to use the ipc_read() and ipc_write() functions for communicating with the task, and the ipc_register() function for handling callbacks.

You can detect the death of the child process by attempting to read from the IPC channel. If you read back zero bytes, then the other end of the pipe has most likely terminated. Alternatively you may use the ipc_register() callback to register notification on the death of a process.

{button See Also, ALink(ipc,,,)}

DDE Communications

DDE is a special type of IPC available only on the Windows platform. DDE is a simple programming language based IPC mechanism, meaning that to implement all aspects of DDE requires low level access to a programming language such as C, or C++. CRiSP supports a sub-set of DDE sufficient to enable CRiSP macros to interoperate with other applications on the system. For example, you can talk to CRiSP from an Excel spreadsheet and write macros which can be called directly.`DDEProgram Manager`

DDE is most commonly used by installation SETUP.EXE programs to ask the Program Manager (on Windows/95 and above, the Explorer) to create program folders and icons.

The semantics of a DDE conversation are usually very different from that of any of the other IPC mechanisms. It probably has more similarity with UDP than with the connection oriented services described in this document. A DDE conversation is transactional: a client application requests some service or data from a server application, usually in a one-off operation. There is not necessarily a permanent connection to a server.

For intimate details of DDE and its internals, you are referred to the official documentation available from most compiler vendors.

CRiSP allows you to write macros which can talk to DDE servers (in which case CRiSP is the client), or to create a DDE server which can receive requests from other applications. A typical client operation would be to send a message to a WEB browser to display a page. Server side operation is useful when you want CRiSP to be notified by some other task of an event, e.g. receive a command to load a file for editing.

Server Side operation

In order to create a DDE server, you need to create an IPC connection using the **IPC_DDE** method, as follows:`Server:DDE`

```
      ipc_create(IPC_DDE | IPC_SERVER, "service:topic");
```

If the operation is successful, then a new IPC id is returned (an integer value) which can be used by the other IPC functions. In order to receive client requests you will need to use the **ipc_register** function.

```
      create_server(string service, string topic)
      {     int   ipc_id;

            ipc_id = ipc_create(IPC_DDE | IPC_SERVER,
                  service + ":" + "topic");
            ipc_register(ipc_id, IPC_TRIGGER_READ,
                  "dde_read_callback " + ipc_id);
      }
      void
      dde_read_callback(int ipc_id)
      {     stringcmd;

            cmd = ipc_read(ipc_id);
            message("Command: %s", cmd);
      }
```

The **service** and **topic** name should be set to something suitable for your application. E.g. the CRiSP DDE server macro (see below) uses "CRiSP" as the service name and "command" as the topic name. Any application wanting to talk to CRiSP has to use these names.

When a client application sends a message to CRiSP, the registered callback routine will be called and the string can be read using the **ipc_read** primitive. CRiSP buffers up commands, although it is best to avoid sending commands longer than 512 bytes otherwise you may find the ipc_read() primitive returning the command string split up, with the first read returning the first 512 bytes, and subsequent reads returning each successive 512 byte fragment until the last fragment is read.

Client Side operation

To make a client request, you need to create an IPC channel and specify the name of the service and topic you wish to connect to. For example to connect to a WEB browser, you would use something like this:

```
int   ipc_id;
ipc_id = ipc_create(IPC_DDE, "NETSCAPE:WWW_OpenURL");
if (ipc_id >= 0) {
      ipc_write(ipc_id, "http://www.crisp.com/");
      }
```

If the specified service is not available then the ipc_create function will fail. Once the connection is open you can use the ipc_write primitive to send commands to the remote server.

CRiSP DDE Server

CRiSP comes with an example macro, **dde.cr**, located in the CRiSP src\crunch directory which illustrates the DDE mechanism and provides a useful mechanism within the editing environment.

This macro sets CRiSP up as a DDE server and allows you to send commands to CRiSP from other programs and have CRiSP execute the commands as if you had typed them in to the **Command:** F10 macro prompt.

This feature is controlled by the **CRiSP Server** option in the **Options**→**Startup** menu.

CRiSP Sample source code

CRiSP comes with some sample source code to illustrate how to talk to a CRiSP DDE server (as established by **dde.cr**). This code is located in the **src\c\dde.c** file and is a utility for talking to CRiSP from the command line.

The tool allows you to specify, on the command line, one or more file names which are to be loaded into a currently running CRiSP session. You can use a command line switch of **+nn** to indicate that the CRiSP is to position the cursor on line **nn** of the next named file on the command line.

A precompiled binary is supplied in the **bin.w32** directory for you to use.

{button See Also, ALink(ipc,,,)}

Using Excel to invoke CRiSP

You can use Excel or any other application which provides access to DDE in order to communicate with CRiSP. This may not be the ideal way to communicate with CRiSP as their are real-time implications of doing this: if CRiSP fails to respond in time for an application, it might time out. However, it can be very useful for an application to be able to communicate with CRiSP, e.g. to pop up a file for editing.`Excel:invoking CRiSP`

This section shows how you can achieve this with Excel, as that is a popular application whose features and mechanisms are shared with other Microsoft Office applications and for which the example is easy to understand.

In order for Excel to communicate with CRiSP, you need to enable the **CRiSP Server** option available from the **Options**→**Startup** menu. This causes CRiSP to create a service (**CRISP**) on startup ready to receive messages from any application.

Excel provides a simple mechanism to invoke a DDE service using the format:

```
=SERVICE|COMMAND|item-name
```

In the case of CRISP, you would specify 'CRISP' for the service name, and 'command' for the command name. The item-name field would correspond to the CRiSP macro you want to invoke, e.g. edit__file.

The return from this *function* is passed back to Excel, but the value may not be of any use - the value depends on the macro you invoke, so you may need to write your own macro wrapper functions to pass back meaningful status information.

Signals Communications

You can use the SIGUSR1 and SIGUSR2 signals under Unix to cause a callback routine to be called. This use of signals is only available under Unix, since Windows does not support a useful signal based mechanism.SignalsSIGUSR1SIGUSR2register_macroREG_SIGNALasync.cr

Signal based communication

The SIGUSR1 and SIGUSR2 IPC mechanism is probably the simplest to use and most primitive way to cause CRiSP to perform an action on demand.

In order to make use of this signal, you need to write a macro which will be invoked when either of these signals is invoked. (CRiSP treats both signals as identical - you cannot distinguish them from within a macro).

To register a macro, use the register_macro() primitive using the REG_SIGNAL trigger condition.

CRiSP contains an example macro which utilizes this feature. Consult the file src/crunch/async.cr in your distribution tree. This macro causes a callback to be invoked when a signal is received. On receipt of the signal, CRiSP looks to see if a file `$HOME/cr.async` exists. If it does, the macro read each line from the file and executes the appropriate macro.

For example, by placing the following command in the file:

```
        edit_menu_file "filename.ext"
```

then CRiSP will edit the specified file. You have complete access to all of CRiSPs macros from within this file. This is a very simple yet very powerful mechanism to cause CRiSP to load a file and display a particular line number.

When CRiSP has finished with the file, it is deleted.

The example macro has definitions in there to also allow you to poll for the existence of the file, rather than relying on receipt of a signal. Using the signals can be a problem in that you need to know CRiSPs process id (PID) before you can send the signal.

The polling timer needs to be configured to a reasonable value (e.g. 1000 milliseconds) before you can use the macro.

There are obvious race conditions in attempting to use signals or polling timers in this manner, but in practise since most people tend to use this facility to automatically open files on demand, and since this occurs fairly infrequently, the race condition can be ignored.

For more sophisticated IPC mechanisms you will need to use any of the other IPC mechanisms.

{button See Also, ALink(ipc,,,)}

Keyboard objects

A **keyboard** is an internal data structure maintained by CRiSP and is used to keep track of the current keyboard bindings. Multiple keyboards can be created and different ones assigned to different buffers, e.g. you could have a set of EDT key bindings in one buffer, vi bindings in another, and compiler specific bindings in yet another.

All keyboards are the same, but you can stack them up in order to overlay or override default bindings for a particular buffer.

A keyboard can be set as the default keyboard, so that the key assignments in the keyboard are applicable to all buffers. Additionally, you can associate a keyboard with a buffer.

The following topics are described in this section

→ Local and Global keyboard

→ Keys and keystrokes

→ Character based terminals

→ Scan codes

→ Creating keyboards and keyboard stacks

→ Keyboard binding files

Local and Global Keyboards

In order to understand what and why keyboards exist, consider what happens when you press a key on the keyboard. When you press a key, CRiSP looks at the current buffer and looks to see if a *local keyboard* has been defined for this buffer. If so, it checks the local keyboard to determine if the current key stroke has a macro binding. If one is found then the associated macro is invoked.

If the local keyboard isn't defined, or if the local keyboard does not contain a binding for the current keystroke, then CRiSP looks at the *global keyboard.* If no key binding is stored in the global keyboard, then the key press is effectively ignored.

This local vs. global behaviour allows you to set up a standard set of key bindings which apply to all buffers but to occasionally define a private local keyboard depending on the buffer.

For example, this mechanism is used by the template mechanism, so that certain keys can be intercepted, e.g. **<Space>, <TAB>,** etc without being concerned with the normal editing keys.

Keys and Keystrokes

CRiSP internally uses a 32-bit code for each key on the keyboard. These 32-bit keycodes are sufficient to allow every key on the keyboard to be encoded and also to handle the modifier keys, such as **Shift, Ctrl, Alt** etc. It is possible to examine these key encodings, but usually it is best to avoid them where possible, since CRiSP provides a high level machine independent way to specify key names. (If you want to know more about the key encoding mechanism, then look at the include file **src/crunch/include/keycode.h**).

In most of the macros and documentation, you will see key name specified inside angle brackets, e.g. **<PgUp>**. This naming convention makes it easy for you to use key names without worrying about the underlying key encodings. These key names are used by various macros, such as **assign_to_key** and allow composite keys to be specified, such as **<Shift-PgUp>, <Alt-Ins>** etc.

Nowadays, most keyboards conform to the standard PC-101 keyboard layout so there is very little variation in the look and feel of any physical keyboard, although some Unix systems do ship with slightly non-standard keyboards.

By using the naming convention, your macros are portable from one environment to another. This is especially useful if you are working in a Unix environment with different vendors keyboards, as the underlying internal codings are different.

As mentioned above, CRiSP uses a 32-bit internal coding for keys. When you specify a keyname, such as **<PgUp>**, CRiSP converts it to the internal key code. CRiSP provides two macros which are useful for converting between the numeric keycodes and the string version (**key_to_int** and **int_to_key**). The **read_char()** macro primitive can be used to read a single keystroke from the user, but it returns an integer value. You can use the above functions to convert to and from the key names.

Character based terminals

In the character based version of CRiSP, life is not as clear cut as it is for the GUI versions when it comes to defining keystrokes. Although CRiSP still provides the various mechanisms documented here for assigning macros to keystrokes, the definition of a keystroke is a little more hazy.

A character based terminal normally contains all the keys you would use in a GUI environment. For example, in an X Windows environment, you use the same keyboard for GUI applications as you do for a terminal command window (e.g. an xterm window). However a character based terminal is limited in the keys that can be identified by an application. Normally it is not possible to determine, for instance that **Alt-B** has been pressed. More than likely the application (terminal emulator) will see the letter **b** or **B** but totally ignore the **Alt** key modifier.

Most terminal emulators are configurable and allow you to program the sequence of characters sent to the application for each key pressed. The default settings are usually of those to emulate a VT-100 style keyboard which is insufficient for CRiSP as it needs access to more keys.

In order to handle the full range of keys on the keyboard, *escape sequences* are usually used to signal specific keys on the keyboard. For instance, when you press the **Up-Arrow** key, three characters are sent to CRiSP: **<Escape>, <[>,** and **<A>**.

In order for CRiSP to know that these three keys represent the single keystroke **<Up-Arrow>** you need to tell CRiSP about these escape sequences.

This is done using the **set_term_characters** macro primitive. CRiSP comes with a set of terminal specific keyboard mappings for various popular terminals. See the macros in the **src/crunch/tty** directory for examples of this.

Scan Codes

As described above, CRiSP uses a set of key definitions which allow you to create macros which are not dependent on the keyboard and system you are using. You do this by using key names such as **<Backspace>**. These key names are translated into key codes. Some keys on the keyboard, notably the keypad and cursor keys are duplicates. (Technically they are not duplicates; whether they are duplicates or not depends on the application or keyboard bindings you have in effect).

To make life easier, when you specify certain keynames, both physical keys on the keyboard are treated together and identically. Thus, you do not need to set up duplicate bindings for the **Cursor-Left-Arrow** and **Keypad-Left-Arrow** keys.

Sometimes this can be just the wrong thing to do. For example, the EDT editor under Digital's VAX/VMS system sets different bindings to the cursor keys and the keypad keys.

CRiSP has an emulator for EDT mode, and in order to work, it relies on using the underlying physical scan codes to make key bindings. Normally, you would use the **assign_to_key** macro and specify a set of key names and macros to bind to the keys. This is not sufficient to distinguish between the cursor and keypad keys. To get around this problem, CRiSP allows you to access the scan codes generated by the underlying system to distinguish similar keys. This is accomplished with the **assign_to_raw_key** macro.

In general, it is advisable to avoid raw key (scan) codes. For examples, the scan codes available under Microsoft Windows are different to the Unix X windows keyboards. In addition, under X Windows, different vendors assign different scan codes to different keyboards, so any attempt to use scan codes is liable to result in failure when operating on different hardware and/or platforms. The CRiSP/EDT emulation has a learning mode dialog to train CRiSP so that the emulation can know the difference between different keystrokes.

In order to use the scan codes, you first need to know what the scan code is for each key. To do this you need to use the **read_char()** and **inq_raw_kbd_char()** primitives (or consult the crisp.log debug output). You then need to convert these scan codes to decimal and use the notation:

"#nnnn"

in the assign_to_raw_key() macro.

Creating Keyboards and keyboard stacks

When CRiSP starts up, there is a default minimalist keyboard defined. This is virtually useless, but acts as a place holder for the user specific key bindings to be loaded into.

The assign_to_key() macro is used to set up bindings for keys, so that CRiSP knows which macro to execute when you press a key.

Sometimes it is desirable to create a private keyboard which has a restricted set of function keys defined. For example, the various character mode popup menus create a popup window on the screen and provide limited key bindings in a controlled fashion, such as **<Up-Arrow>, <PgDn>, <Home>** etc. Rather than *hide* all the default key bindings, a new keyboard is created which overrides the current global one.

This is done via the **keyboard_push()** primitive. This creates a brand new keyboard, which has no key bindings in it and saves the previous global keyboard on a stack for later retrieval. Having called **keyboard_push** you can now call **assign_to_key** without affecting the existing global key assignments.

When you are finished with the keyboard, you call **keyboard_pop** to discard the keyboard and throw away all the key bindings you have made. The **keyboard_pop** primitive takes an optional integer argument. By default the keyboard is discarded, but you can specify a value which does not discard the keyboard, but simply removes the keyboard from the current list of keyboards to be scanned when a key is pressed.

This allows you to create a keyboard, for instance in the **main()** part of your macro, and keep it to one side until needed, e.g. when you popup a window.

Various primitives are available for populating a keyboard:

copy_keyboard
This allows you to copy bindings from another keyboard into the current one. For instance, you might want to copy the macros which are assigned to the arrow keys. This would allow your macro to use the users preferential arrow keys, e.g. if they are using vi or CUA mode.

keyboard_typeables

This populates the current keyboard with all the normal typeable characters (A..Z, a..z, 0..9, etc) so that they are self-inserting (see **self_insert**).

keyboard_reset
Clears out all key bindings from the current keyboard.

Keyboard binding files

CRiSP provides a facility for storing keyboard bindings in a file separate from a macro. The CRiSP distribution comes with variety of editor emulations, which are stored in these files. (See **etc/*.kbd** in the distribution).

These files are designed to allow you to select a base emulation, and then allow you to modify or enhance these basic definitions. This facility is provided by a set of macros.

The primitives to enforce this are: **set_kbd_name, inq_kbd_name** and **load_keyboard_file.**

At any one time, CRiSP remembers the current editor emulation, e.g. **Vi, Emacs, EDT.** This allows CRiSP to load the appropriate keyboard file on startup, and makes it easier for the user to select the bindings from the setup menu. This current keyboard emulation name is set via the **set_kbd_name** primitive and can be retrieved with the **inq_kbd_name** function.

The **load_keyboard_file** function allows you to define a set of keystrokes *en masse* CRiSP without having to code a set of assign_to_key macro calls.

CRUNCH: Things to watch out for

1. Crunch does not generate code for old style C declarations. The new style must be used.
2. Crunch does not verify that the return value from a function agrees with the return type declared for the function. (It does check return(expr); and return; statements against whether the function is void or not).
3. Crunch does no compile-time checking of the reasonableness of expressions, regarding types, For example:

   ```
   list    l1, l2;
   ```

   ```
   l1 = l1 * l2;
   ```
4. Crunch complains about unused arguments sometimes because it cannot intuit whether an argument is going to be used via the dynamic scoping rules. A language extension or lint-style comments are needed to tell the compiler not to complain about this.
5. If a local variable is declared at the top level of a function definition with the same name as a parameter to the function, the user is not warned and possibly incorrect results will occur.
6. Case statements which flow into each other are not interpreted the same as C.
7. When writing *replacement* macros to overload the functionality of built-in macros there can be a name scoping problem. For example if you define a macro called **printf()** and use a parameter **fred** as a local variable inside that function then CRiSP may access the wrong variable if fred is defined in the function which calls **printf().**

The cm compiler

CRiSP includes a utility called **cm** which is a low-level compiler and disassembler for CRiSP. The .m language may be considered as the assembly language of CRiSP, and looks and feels like the Lisp language. CRiSP internally executes a Lisp-like execution engine and the .m language maps directly onto this language. The normal C-like crunch language is compiled, using the crunch language, into a compiled version of the .m code. This results in faster execution.

The major drawback of the .m language, and hence the reason it is no longer in use, is that it is very difficult to read and write by a human being, because of the plethora of brackets used to enclose execution statements.

The crunch compiler can be used to generate .m code by use of the command line -c switch. (Normally .cr files are compiled directly into loadable files with a .cm extension).

CRiSP can load .m files directly without any need for compilation, but this can be slower than loading an equivalent compiled file so this facility is rarely used or needed.

The .m source files are compiled with the **cm** program.

***** NOTE *****

> The name '**cm**' conflicts with the calendar manager utility which is available under SunOS 4.x and Solaris 2.x when using OpenLOOK. Because of this conflict, you may have problems accessing the correct 'cm' program when you try to run this program from within XView. You can use the crunch compiler to compile '.m' files and this knows how to avoid running the wrong 'cm' program so it is advisable to use this.

The **'cm'** compiler converts macros in source form (.m files) to a compact compiled form (.cm) files. CRiSP can read .m files or .cm files at run time, but the .cm format loads faster and avoids CRiSP having to parse an input file, skipping over comments etc.

The command line syntax for 'cm' is:

```
cm [-aLl] [-o output_file] file1.m file2.m ..
```

The usual way to invoke **cm** is simply:

```
cm fred.m
```

This compiles the file fred.m, and creates a file fred.cm in the same directory as fred.m. Compiled macro files are loaded significantly more quickly and execute slightly more efficiently than pure source files.

cm checks for syntax errors as it goes along - the most common errors are unmatched brackets, and unterminated string constants.

cm has a number of switches. Most of these are used for debugging the compiler itself and for disassembling compiled macro files.

-32, -64	CRiSP supports two types of macro file formats, one for 64-bit machines and the other for 32-bit machines. You can use these switches to convert a .cm file into the appropriate format. Not normally needed by end users.
-a	This switch is used to print out the relative amount of space used by individual components of the compiled language. It is used for fine tuning the pseudo code generated by the compiler. Most users can ignore this switch.
-l	This switch is used to print out each macro so that the internal parsing can be checked. This switch currently does not work.
-L	This switch is used to print out the internal pseudo code used to represent a macro and allows debugging of the internal code generator. This switch currently does not work.
-o filename	Specify name of output file to create.
-S	Dump the macro string table section.
-V	Print the version number.