



University of Amsterdam
Faculty of Mathematics,
Computer Science, Physics
and Astronomy

TNO Institute of
Applied Physics

Kruislaan 403
1098 SJ Amsterdam,
The Netherlands

P.O. Box 155
2600 AD Delft, The Netherlands
Stieltjesweg 1
2628 CK Delft, The Netherlands

SCIL_Image

version 1.4

User Manual

WINDOWS version

May, 1998

Copyright notice

Copyright © 1992-1998 by University of Amsterdam, Faculty of Mathematics and Computer Science, Amsterdam, The Netherlands and TNO Institute of Applied Physics, Delft, The Netherlands. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the prior written permission of TNO Institute of Applied Physics, Delft, The Netherlands.

Disclaimer

It is believed that the information in this publication is accurate as to the date of publication; this information and the software package, which is described, are subject to change without notice. Furthermore, University of Amsterdam, Faculty of Mathematics and Computer Science and TNO Institute of Applied Physics make no representations or warranties as to the accuracy or completeness of this publication, nor as to the accuracy or completeness of the software-package it describes. All other warranties, express or implied, are hereby disclaimed, specifically including, but not limited to, express or implied warranties of merchantability or fitness for a particular purpose.

Chapter 1 Outlines of SCIL_Image	1-1
The four layers of SCIL_Image	1-2
Using the system at the application level.....	1-2
Using the system as an interactive user	1-2
Using shorthand typing for application development.....	1-3
Control statements	1-3
Programming an image processing function.....	1-4
Making your own library	1-5
Building applications	1-5
SCIL and the Image library.....	1-5
On reading this manual	1-6
Summary of the Chapters.....	1-6
Why SCIL_Image ?	1-8
A multi-level interactive processing environment.....	1-10
SCIL: Library handler.....	1-10
SCIL: C-interpreter	1-11
SCIL: Command expander	1-11
SCIL: Menu and dialog generator.....	1-12
The Image libraries	1-12
Image infrastructure	1-12
Binary mathematical morphology.....	1-13
Numerical analysis of objects in images.....	1-13
 Chapter 2 Setting up SCIL_Image	 2-1
Setting Up: Step by Step.....	2-2
The SCIL_Image Folder	2-4
Your SCIL_Image Environment.....	2-6
 Chapter 3 Getting started.....	 3-1
Before you begin.....	3-2
The Five Modes of Interaction with SCIL_Image.....	3-2
Session One: Viewing Images	3-3
Session Two: Using the SCIL_Image Menu System.....	3-13
Session Three: Using the SCIL_Image Command Line Mode	3-21
Session Four: Programming in SCIL_Image.....	3-26
Making a New Compiled Version of SCIL_Image	3-30
The Commands of the SCIL_Image Menus	3-37
File	3-37
Edit.....	3-37
SCIL.....	3-38
Image.....	3-38
Display	3-40
Options.....	3-40
Arithmetic	3-41
Itools	3-41
The Properties of Text Windows	3-42
On-line manuals	3-43
Reference manual.....	3-43
 Chapter 4 The C Interpreter	 4-1
SCIL_Image and C	4-2
ANSI-C compatibility.....	4-3

The Direct Command Mode	4-3
The Macro Mode.....	4-5
The Programming Mode: Interpreted C-functions and UFOs	4-7
Program Development Commands	4-9
chain <filename> [args]	4-10
list [start],[end].....	4-10
load <filename>	4-10
logon <filename> logoff	4-11
macro [-i] [-v] <macrofile>.....	4-11
rmvar	4-12
run [args].....	4-12
time <command>	4-12
the interrupt: Pause/Break.....	4-13
help facilities:.....	4-13
Ctrl-H <selection>Ctrl-Enter <command> ? ? <pattern>.....	4-13
Errors, Warnings and Diagnostics	4-15
Features of SCIL_Image s C-interpreter.....	4-16

Chapter 5 Advanced SCIL_Image..... 5-1

Adding New Functions to SCIL_Image	5-2
Making a Command Description File (CDF)	5-3
comment entry	5-4
menu entry	5-4
translate entry.....	5-5
variable entry	5-5
command entry.....	5-6
SCIL_Image Special Types	5-10
Very Advanced SCIL_Image: New Types	5-14
Creating a New Compiled SCIL_Image Version	5-16
Adding On-line Manual Files to SCIL_Image.....	5-17

Chapter 6 The Image 2.1 library in SCIL_Image 6-1

Introduction.....	6-2
Image infrastructure	6-3
Invalid operations.....	6-3
Image display and window management.....	6-4
Mouse buttons.....	6-4
The title bar of image windows	6-4
The left mouse button	6-4
Displaying the image	6-5
Displaying 3D images.....	6-6
The right mouse button	6-6
Changing a window s size.....	6-7
Changing a window s position.....	6-7
Display lookup table	6-7
Image management.....	6-8
Creating and destroying images.....	6-8
Changing image sizes	6-9
Changing image types.....	6-9
Converting images into other types	6-9
Filling images	6-10
Region of interest (ROI) processing	6-10
Image types	6-11
Grey valued images (GREY_2D & GREY_3D)	6-11
Data representation of grey valued images.....	6-11

Usage of grey valued images	6-12
Examples of grey valued operations	6-12
Binary bitmapped images (BINARY_2D & BINARY_3D)	6-12
Data representation of binary images	6-12
Usage of binary images.....	6-13
Examples of binary operations.....	6-13
Floating point images (FLOAT_2D & FLOAT_3D)	6-14
Data representation of float images	6-14
Usage of float images.....	6-14
Examples of operations on float images	6-14
Complex images (COMPLEX_2D & COMPLEX_3D).....	6-15
Data representation of complex images	6-15
Usage of complex images	6-15
Examples of operations on complex images.....	6-15
Labeled images (LABEL_2D & LABEL_3D)	6-16
Data representation of labeled images	6-16
Usage of labeled images	6-16
Examples of operations on labeled images.....	6-16
Color images (COLOR_2D & COLOR_3D)	6-17
Data representation of color images	6-17
Usage of color images.....	6-18
Expression evaluation on images (eval)	6-19
Storing images on disk.....	6-22
The ICS format	6-22
The TIFF format	6-22
The JPEG format.....	6-23
The TCL format	6-23
The AIM format.....	6-23
Non image data (var_objects)	6-24
Behavior of var_objects	6-24
Datatypes of var_objects.....	6-25
Examples using var_objects.....	6-25
Example 1	6-25
Example 2	6-26
Example 3	6-26
Storage of var_objects on disk.....	6-26
Histogram objects	6-26

Chapter 7 Introduction to Image 2.1 **7**

What is Image2.1	7-2
Image infrastructure	7-2
Image types	7-2
Advanced and extensive set of image operations	7-3
Fast implementations	7-3
Abstract error and I/O handling	7-3
Publish and subscribe mechanism	7-3
The structure of this manual	7-4
The need for Image2.1	7-4
Structure of the Image library	7-5
Writing your own image processing routines	7-5
Custom Image types.....	7-6
Measurement of objects (AIO)	7-6
Binary images	7-6
Appendixes	7-7
Appendix: ICS file format description.....	7-7

Chapter 8 Publish and Subscribe	8-1
General aspects	8-2
Object requirements	8-2
Subscribing and unsubscribing to objects	8-3
Receiving messages	8-3
Publishing messages	8-4
Processing messages	8-4
Messages	8-5
Publish and Subscribe in the Image library	8-5
Top-level publishes	8-5
Image publishes	8-6
Color-lookup table publishes	8-7
Top-level Histogram publishes	8-8
Histogram publishes	8-8
Error stack publishes	8-9
Chapter 9 Programming with Image	9
Introduction to Image	9-2
The Image types	9-4
Grey valued images	9-4
Binary bitmapped images	9-4
Floating point images	9-4
Complex images	9-5
Labeled images	9-5
Color Images	9-5
The IMAGE structure	9-6
Image Flags	9-7
Region of interest data structure (rectangular)	9-8
Region of interest data structure (arbitrary shaped)	9-9
Operation Counter	9-9
Image Info	9-10
Image Color Lookup Tables	9-11
Dynamic adjustment (Pre_op, Post_op)	9-12
The pre_op function	9-14
COMPARE mode	9-15
ADJUST(_NIP) mode	9-15
Output equal to input	9-16
Output of specific type	9-16
Only output	9-16
Type of input, sizes of output	9-17
Special sizes	9-17
ROI and pre_op	9-18
Multiple calls to pre_op	9-18
The post_op function	9-19
ROI and post_op	9-19
A simple example	9-19
Explanation of the function code:	9-20
Error handling and reporting	9-21
Location of the error	9-21
Return values of functions	9-22
Error handling	9-22
Checking routines	9-23
Check functions	9-24
Check_image_integrity	9-25

Textual output	9-26
Function overloading	9-27
Three layers.....	9-27
Generic function layer.....	9-28
Parameter checking and image adjustment.....	9-28
Processing the data.....	9-29
Overload tables	9-30
Overruling the default implementation.....	9-31
Data conversion (convert).....	9-32
Super type of an image line	9-32
COMMON_LINE structure	9-32
Source function specification.....	9-33
Destination function specification	9-35
User specified conversion	9-37
Var_objects	9-37
Var_object structure.....	9-37
Programming with var_objects	9-39
Checks on var_objects	9-39
Conversion of var_objects to images and vice versa	9-39
Histogram structure.....	9-40
Programming with histogram objects	9-42

Chapter 10 Analysis of Images and Objects (AIO)..... 10-1

General Concepts in Microscopical Image Analysis.....	10-2
Components of the AIO framework	10-3
Labeling objects.....	10-3
Measuring individual objects.....	10-3
Object manipulation.....	10-4
Image Silo	10-4
Direct manipulation	10-4
An AIO sample session.....	10-4
Interactive measurement	10-6
Implementation of the interaction part.....	10-7
The point_object() function	10-7

Chapter 11 Bitmapped binary images..... 11-1

Erosions, Dilations and Logarithmic Decomposition.....	11-2
Algorithmic Implementation.....	11-4
Data Representation	11-4
Implementation of the Pixelwise Logical Operations.....	11-6
Implementation of the Morphological Operations.....	11-7
Evaluation	11-8
Pixelwise Logical Operations	11-9
Morphological Transforms	11-10
Discussion and Conclusions	11-10
Literature.....	11-11

Chapter 12 New image types

Implementing a new image type	12-2
Defines and structures.....	12-3
Creation and destruction	12-4
Copying a part of the image.....	12-6
Displaying the image	12-8
Image type information.....	12-11

Conversion to other image types.....	12-12
The overload table.....	12-14
Low Levels.....	12-15
Testing the image type	12-18
Defining an image subtype	12-19

Chapter 13 Creating an Image Application..... 13-1

Initialization	13-2
Error handling	13-2
Text output handling	13-3
Color Lookup tables.....	13-3
Sample code.....	13-3

Chapter 1 Outlines of SCIL_Image

In this chapter an outline of SCIL_Image is given. Because SCIL_Image is a multi-layered and multi-purpose system, this may help you in orienting yourself in the system's use.

Read this chapter if:

- You are using the system as an interested user of image processing applications,
- You are using the system to develop image processing routines or applications or,
- You are to install the system, or,
- You want to understand the SCIL_Image philosophy.

Do not read this chapter if:

- You are using an application based on SCIL_Image, and SCIL_Image and the application have been properly installed for you. Read the application manual and in a later stage read this manual if necessary.

The four layers of SCIL_Image

SCIL_Image is an extensive multiple layered system for image processing and for the development of applications in the image processing domain. As a consequence of the layered structure, SCIL_Image can be operated by users with various levels of expertise with image processing and with various levels of programming skills: from a user of an image processing application to an image processing system developer.

Using the system at the application level

An image processing application is a set of image processing commands or a tailored program made available to the user, usually in the form of a menu option. It is possible that the standard image processing operation of SCIL_Image are still available to the user, but sometimes they are not. This is not the choice of the SCIL_Image team, but rather a choice of the developer of that specific application. Using the system at the application level boils down to selecting options with the mouse and filling in parameter values where needed. The manual of the specific application is not a part of this manual, as this manual describes only the generic use of the SCIL_Image system. To get an overview of the standard SCIL_Image system, the reader is referred to the chapter "Getting started" on page 3-3-1. for introductory sample sessions. The remainder of this manual is about using SCIL_Image for interactively analyzing images using the extensive image processing library of SCIL_Image, for building image processing applications.

Using the system as an interactive user

At the highest level of use, functions can be executed one by one by interactively selecting them from the menu system. All operations of SCIL_Image are available to the interactive user ranging from reading an image, displaying the image to enhancing the contents, making a histogram or a Fourier spectrum, performing mathematical morphology, and so on. A quick overview of the functions is given in the guided tour through the system, see "**Getting started**" (chapter 3). More information on the image processing functions is given in "**Introduction to Image 2.1**" (chapter 7) and further. The best way to get acquainted with the system is, however, to use and try functions on your images.

To enhance the flexibility of the system, functions have parameters and options to be filled in by the user. A parameter may be the symbolic image name, a code to change the behavior of the operation, a value used as a selection criterion, etcetera. In selecting the image processing functions from the menu, the user is assisted by the system in the selecting parameter values.

Usually a default value is available which should normally do the job. Also, with almost all functions, help is available giving a written explanation to the purpose and method of the function and its parameters.

Using shorthand typing for application development

When a menu option has been selected it is translated by SCIL into a command. The command appears in the text window. Subsequently, it will be executed. In SCIL_Image, any menu selection is automatically translated to a written command, inserted in the text window and executed only then. This is done to serve the user as he or she becomes more experienced with the system and the image processing libraries. More experienced users may find interaction via the menu system tedious. They rather type commands by their name, rather than selecting them from the menu. In SCIL_Image, a user may type the commands in the text window directly. As all menu commands are translated into written commands, typing commands may be mixed with selecting and executing items from the menus. There is no difference. Using the menu system is easier to learn and there is more support to the user in selecting parameter values and options, but typing commands is quicker. Therefore, simultaneous use of the two interface levels combines the best of both worlds.

Control statements

At a point, connecting image processing commands may no longer be good enough to reach your image processing target. Frequently, streams of image processing commands are combined into a new, high level image processing operation. Or, a set of commands is executed as many times as there are particles in the image. Or, some stream of image processing commands is repeated to improve the quality of the image until it cannot be improved any further. Or, an image processing measurement function results in a numerical value (e.g. the image contrast, or the number of particles in the image) and that number is needed as a parameter value in another operation (e.g. an operation to normalize the contrast or an operation to select all particles with a certain minimum area).

If such a control over the flow through the image processing commands is desired, control statements are needed. These statements cannot be selected from the menu, they can only be typed in the text window. The SCIL_Image system uses the syntax of the C programming language in specifying the control commands. A control command is typed in the text window in the standard C syntax. Upon completion by the <return> or <enter> key, the statement is interpreted and executed. The C interpreter will check for syntax, availability of the requested function and then issue the requested function. For an example, see the guided tour in "Getting started" on page 3-3-1.

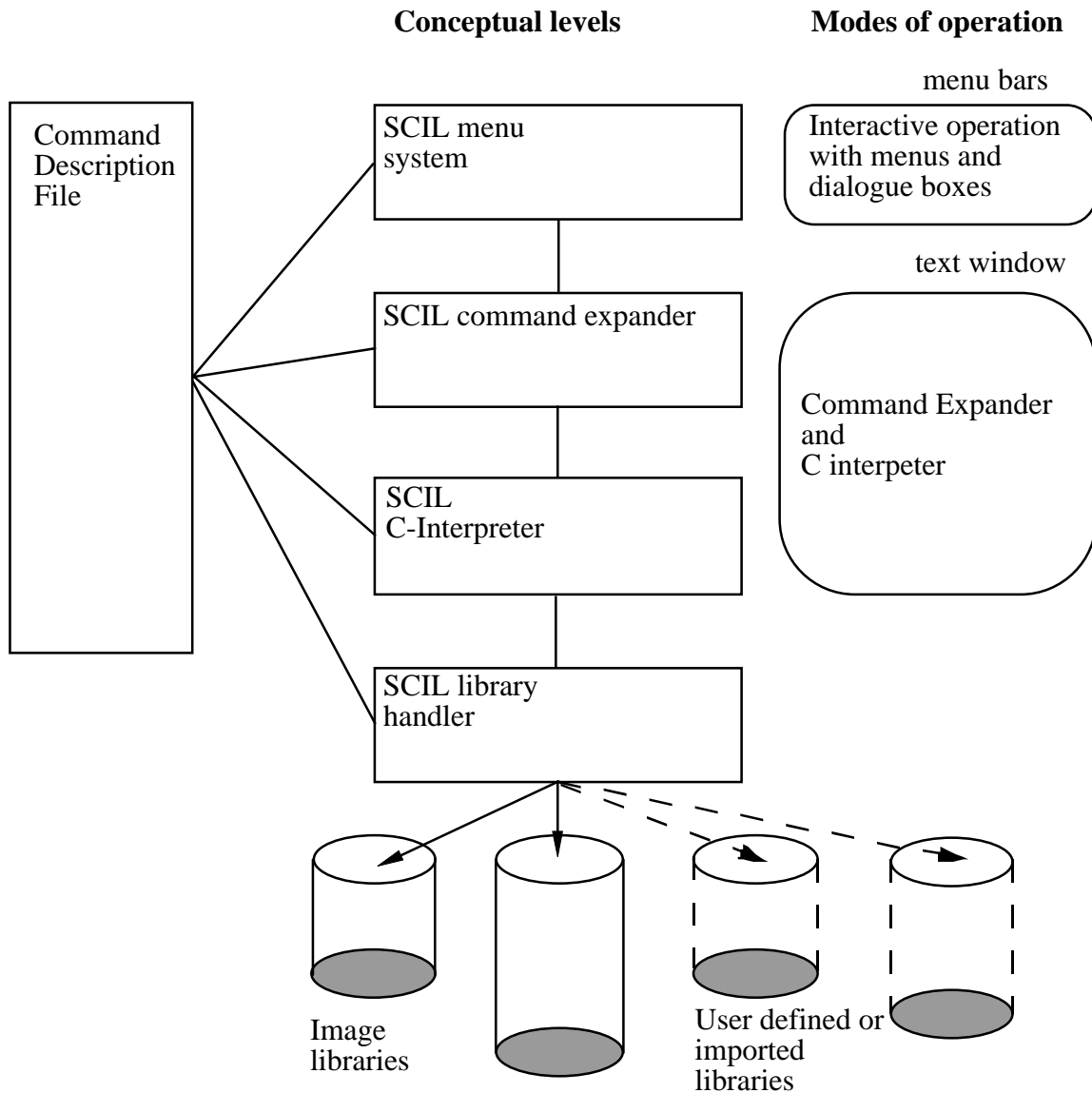


Figure 1-1 : The layers of SCIL_Image and its modes of operation

Programming an image processing function

Going one step further in expertise, the need may arise to developed a new image processing function. This can be done by programming at the level of the C interpreter. Newly developed functions can be made available quickly in the SCIL_Image system by inserting the necessary function identification information in the Command Description File. Restarting the system is sufficient to integrate your new interpreted function with standard SCIL_Image. The function will appear in menus and is ready for use.

The Command Description File contains the layout of the menu-system and on each available command; the information needed by SCIL_Image to execute that command properly. Modifying the Command Description File is appropriate is you wish to hide some (or the

majority) of the Image functions from your users' sight. Or, may you wish to re-order the image processing functions differently over the menus, more suited to your specific needs. This can all be done by editing the Command Description File.

Making your own library

When a algorithm has been fully developed and tested, the set of commands which constitute the algorithm used can be converted to C-routines with little effort. In general, this only requires the addition of a subroutine header and tail. A new routine can be added to SCIL_Image by compiling it with a standard C compiler and inserting the routine in your own library. At the same time, the name of the function and the parameters and the admissible parameter values must specified in the Command Description File to make it acquainted to the system. When this has been done correctly, the function will be available to the user in the same way as any of the standard image processing functions.

Building applications

For a specific application it is often felt desirable to provide the user access to only those menu items which are meaningful for that particular applications. To accomplish this, the user interface of the SCIL_Image system can be altered at will by adding, removing and re-ordering menus and menu items.

SCIL and the Image library

Although from the outside SCIL_Image looks like a fully integrated environment, in reality it is constructed from two separate entities, SCIL and Image. SCIL being the C-interpreter and menu & dialog generator. And Image (the Image 2.1 library) being the Image infrastructure and a large set of image processing functions. These two parts, supplemented by a user interface for the visualization of the images make up the SCIL_Image package.

The Image library can be used separately to create stand alone (end-user) applications. This means that after trying out some new ideas and gradually developing image processing functions in the full featured SCIL_Image environment, they can be used at once in these stand alone applications.

On reading this manual

This is version 1.4 of the SCIL_Image manual. To work with SCIL_Image it is not necessary to read the entire manual, neither is it necessary to have experience with the programming language C, even though the system is based on a C interpreter.

When using the SCIL_Image system for command execution only, it is sufficient to know about the menu system and the command level. Selecting an item from a pull down menu will result in interaction with the system via a dialog box and subsequent execution of the command. Reading Chapters 1 through 3 should be sufficient for this level of use, although Chapter 4 may also be useful. On the other hand, if the system is used for the development of new routines and applications, it will be necessary to become familiar with the system's philosophy and C. In fact, the step from a sequence of commands typed in the window to a complete C-program is small. Advanced use of C and the image data structures may boost development greatly. When developing algorithms and applications, reading Chapters 4 and beyond is recommended.

Summary of the Chapters

Part I of the manual provides a general orientation:

- Chapter 1** gives a global description of the SCIL_Image system.
- Chapter 2** explains how users should set up their personal environment and obtain a personal version of SCIL_Image.
- Chapter 3** describes how to start using the system and presents *sample sessions* which allow users to get some experience using the system. The chapter ends with a discussion on some general design issues, such as the *look and feel* of the user interface.

Part II is about the interaction system SCIL and Image as part of SCIL_Image.

- Chapter 4** describes the C interpreter SCIL. It explains the *direct command mode*, the *program mode*, the *history mechanism*, the *command expander*, the *line editor* and various SCIL commands.
- Chapter 5** describes advanced use of the *C interpreter*, including how to add new routines to the system, and how to modify the *user interface*. This is particularly useful for application developers.

Chapter 6 deals with the image processing of Image as part of the SCIL_Image environment. The philosophy and general behavior of the image processing infrastructure and user interface are discussed

Part III is the programmers manual of the Image 2.1 library.

Chapter 7 is an general introduction to the Image 2.1 library and to the manual.

Chapter 8 deals with the publish and subscribe message passing mechanism. Image 2.1 uses this to communicate with any interface.

Chapter 9 focuses on advanced use of Image, particularly useful for the program developer of image processing functions.

Chapter 10 describes the AIO-library.

Chapter 11 handles the philosophy and use of the binary bitmapped routines.

Chapter 12 describes how to implement new image types.

Chapter 13 is about writing an application with the Image 2.1 library

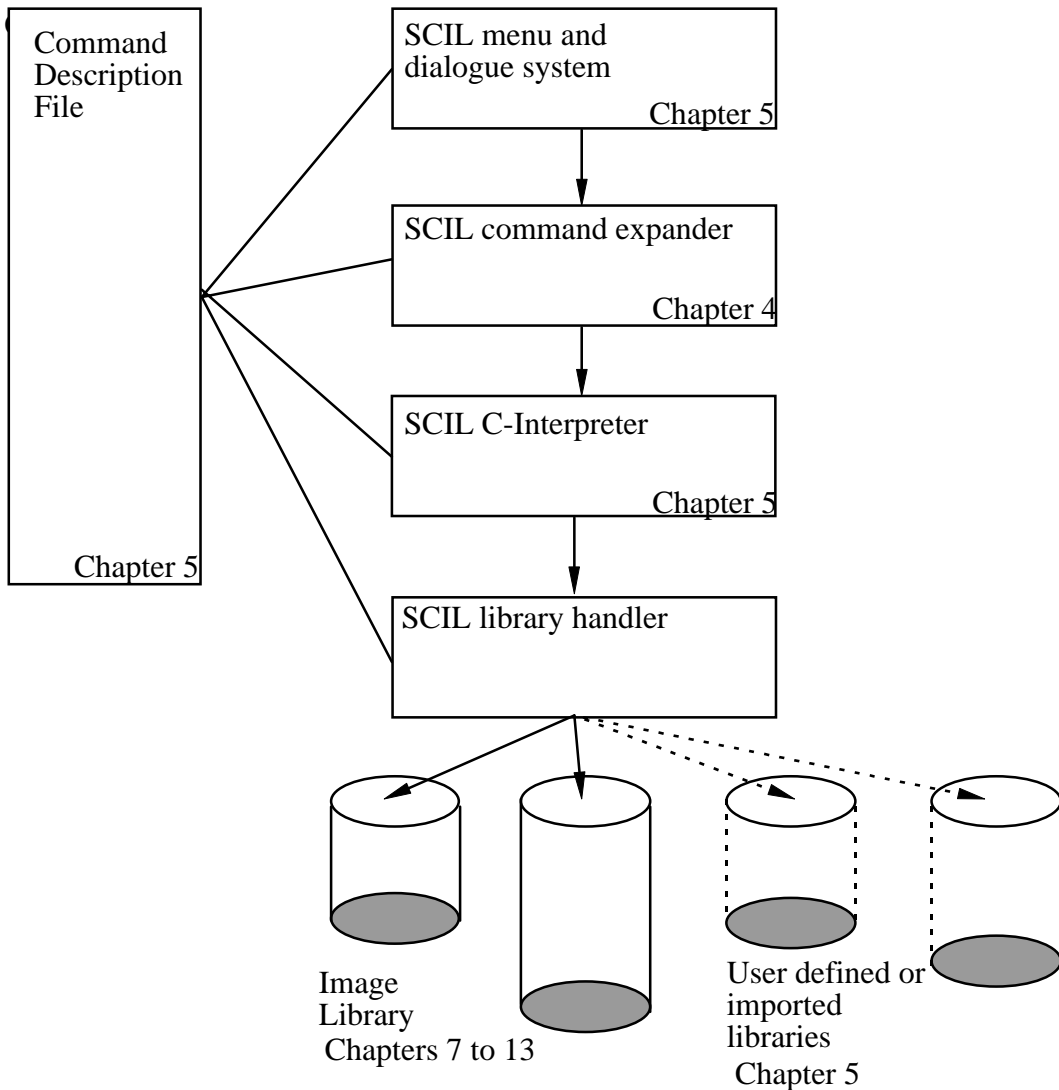


Figure 1-2 : An index into the SCIL_Image manual

Why SCIL_Image ?

Interactive environments have always played an important role in image processing, because of its visual nature. Generally, there are no off the shelf solutions available for image processing problems. The visual evaluation of image processing techniques helps to find solutions to these problems. The immediate feedback interactive systems provide reduces the time necessary to develop new applications. Furthermore, the ability to see the results of commands and to modify code or parameters within seconds brings new insights, for example as to how sensitive an algorithm is to a small change in the image.

The natural alliance between interactive systems and image processing has led to the development of an enormous variety of software systems for image processing, most of which are bound to specific hardware or tailored to a specific image processing application domain.

However, Preston concluded in his 1981 survey covering 72 image processing languages, that the diversity in image processing environments is not justified since image processing applications are not so disparate as to require unique and specialized data types or different programming constructions. Rather than creating a new highly specialized language it is better to make use of an existing programming language.

The functionality of existing programming environments range from a collection of library routines, via command driven systems or menu driven systems, to highly specialized programming languages.

In the first category of systems each of the routines comprise a command. No environmental support is provided. Although this category offers the advantage of development in a standard programming language, these systems are less suited for interactive image processing and are, in general, not user-friendly.

Command driven interactive systems or command interpreters are characterized by prompt execution of image operations. The functionality of command interpreters ranges from those which offer the execution of operations and the control of parameters to those which provide variable declaration, expression evaluation, flow control and/or procedure execution. It is important that command interpreters provide tools for the constructions of new commands composed of a sequence of basic commands. Although systems in this category are geared towards interactive processing most do not have the flexibility and complexity inherent in a standard programming language. Moreover, the diversity of semantics and syntax between interpreters obstructs user acceptance. An additional disadvantage of command interpreters is that the names of commands and the sequence and meaning of the parameters must, in most cases, be known by heart.

In contrast, menu-based interactive systems allow easy access to image processing commands. By the menu system, help can be offered to the user, as well as default values of parameters can be suggested. The menu system provides a user-friendly interface suitable for novice users.

Based on the above discussion, we believe an image processing environment should fulfill the following requirements:

- The system should contain an extensive set of image processing routines, readily available to the user.
- Visual feedback must be given in immediate response to selected or specified commands, at least the ones which modify the image.
- A menu system and command interpreter should be available simultaneously to support novice and advanced users.

- It should be possible to both develop and use applications within the same environment, to escape the gap between program development and use.
- One package should be made available on a variety of machines rather than separated ones developed for each machine type. This includes the window system and the command language.
- It should be possible to execute commands specified in a shorthand form.
- The image processing functionality of the system should be extensible both with private routines as well as routines from other packages accessible as a subroutine library. As an example, consider extending SCIL_Image with an interface to a database package, a spreadsheet or a statistical package.

A multi-level interactive processing environment

SCIL_Image has been developed as a multi-level environment. Figure 1-3 shows the internal levels of the environment.

SCIL: Library handler

To provide expansion of the system, software libraries are linked via the library handler. The system comes with quite a few libraries for image processing functions. Through the Library handler, the system can be expanded with external libraries or with any other user provided software library. The data structures to get access to the image or parameter data as in use in SCIL_Image are documented in "**Programming with Image**". Compiled routines can be integrated in the system by adding an entry to a Command Description File.

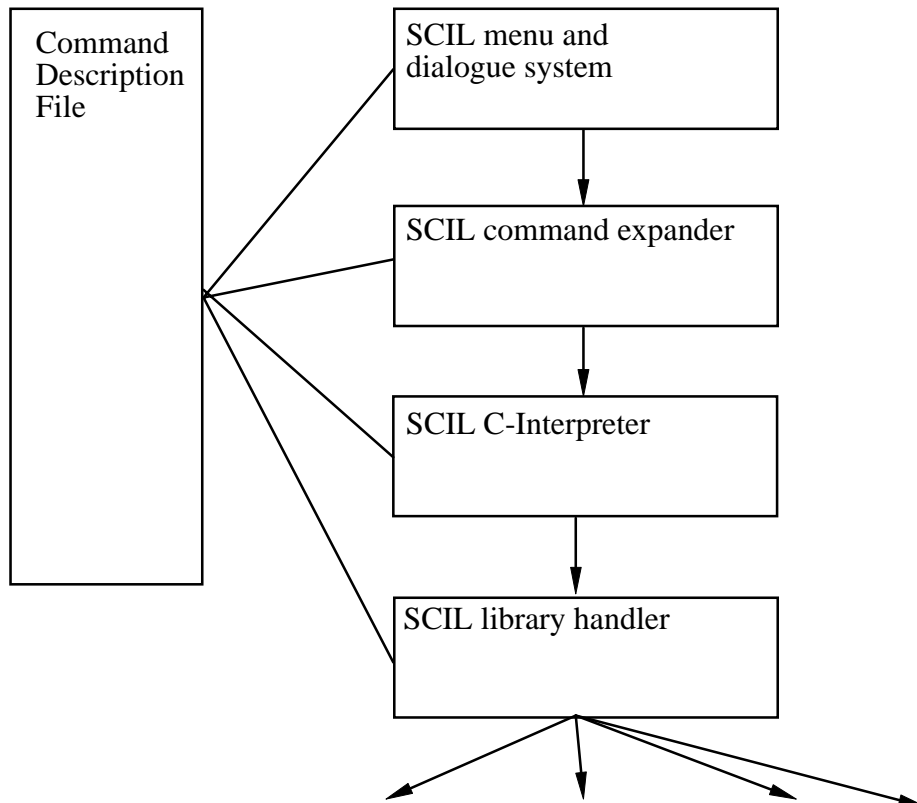


Figure 1-3 : The multi-level interactive environment SCIL.

SCIL: C-interpreter

Using SCIL, a C interpreter for command interpretation ensures general programmability since algorithms are developed with the full capabilities and flexibility of the C programming language. An additional advantage is that because development takes place in an interpreter, using standard C syntax, the source code can be compiled and incorporated in the system, with the use of the library handler.

SCIL: Command expander

The basic task of the command expander is to translate shorthand commands entered by the user to a complete C statement. This is to avoid the user being bothered with the precise typing of the C punctuation whenever possible. Simultaneously, the command expander checks the arguments of functions for consistency and, if any are missing, fills in default values. The command expander operates according to a Command Description File (CDF), in which the information on each command and its arguments is stored.

SCIL: Menu and dialog generator

The menu-and-dialog generator provides an interface that makes it possible to select commands via the mouse rather than by typing them. The interface is generated when SCIL_Image is started based on specifications in the Command Description File. When the user selects a menu option, a dialog box is generated. The box gives the parameters of the command with their defaults and ranges. The graphical presentation of the items in the dialog box is adjusted by the type and range of the parameters. In SCIL_Image, the menu-and-dialog generator performs an educated guess to produce the most convenient dialog.

The Image libraries

The Image libraries offer a large variety of image processing tools. Figure 1-4 gives an overview of the libraries available in Image. As a user, you will not notice the existence of different libraries because the library handler and the Command Description File make every function known to the system in the same way.

A large variety of filter and image manipulation routines are available. These include arithmetical operations, Fourier transformation, geometrical routines, and image measurement. A number of these have been adapted from the well known TCL-Image package. There are too many functions to discuss here, we only give a few examples. A quick overview is given in the guided tour in "Getting started" (chapter 3).

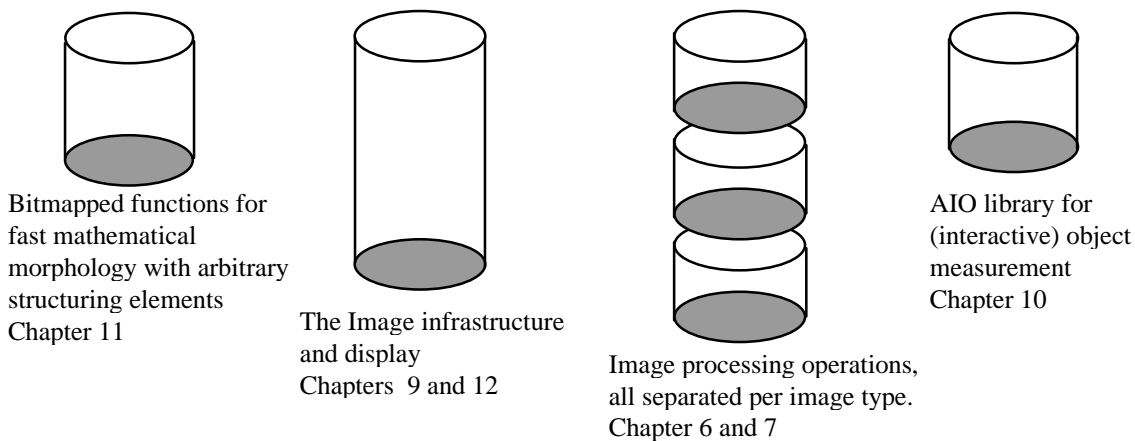


Figure 1-4: The Image libraries.

Image infrastructure

Image is based on a division of the images and image processing routines by type. This type mechanism simplifies the management of images, and makes programming with different types of images easy. Each type has its own functions for type dependent tasks, that are accessed by function overloading. Currently available types in both 2D and 3D are: binary,

grey value, float, complex, label and color. The image infrastructure is particularly helpful in maintaining a fail-safe program when programming in an extensive image processing system. When you are an advanced programmer of SCIL_Image, you may wish to define your own image types. You can do so, but it is advised only to do so when you have the proper experience. See Chapter 12 for details.

Binary mathematical morphology

A complete and fast implementation of mathematical morphology routines is available. These include routines for erosion, dilation, closing and opening, using arbitrary sized and shaped structuring elements. These are optimized implementations for circular structuring elements. Grey value morphology operations are also included in the Image libraries.

Numerical analysis of objects in images.

When images consist of a set of small objects, such as microscopical images of cells and grains, or local details, the AIO-library provides facilities to measure object features. Measurements can be made in automatic or interactive mode. The list of such feature values is stored in an editable file. Such a file may be converted (with the aid of an editor) into a file suited for a standard statistical package.

Chapter 2 Setting up SCIL_Image

This chapter describes how to set up SCIL_Image. It tells you how to install the SCIL_Image software, and how to set up your working environment.

Read this chapter if:

- You are setting up SCIL_Image.
- You would like to change your working environment.

Do not read this chapter if:

- SCIL_Image is working to your satisfaction.

Setting Up: Step by Step

In this section, we assume that you have just received your SCIL_Image package. That package consists of:

- A CD-ROM with the SCIL_Image package.
- A Hardware Protection Key (dongle).
- A License Registration Form (containing the license code string).

You need the following to run SCIL_Image:

- A 486/DX computer, 33 MHz or better.
- At least 12 Mb of memory
- A 1024x768 monitor with 256 colors (recommended).
- A hard disk with minimally 15 Mb of free space (recommended).
- Windows 95 or Windows NT 4.0.
- Microsoft Visual C++ compiler (only if you wish to extend SCIL_Image's functionality).

The installation procedure is as follows:

- 1) Insert the CD-ROM in your CD-ROM drive and start the installer (**setup.exe**) by double clicking its icon:



- 2) The installer (the workings of which should be self-explanatory) will install SCIL_Image in a directory of your choice. When finished, the **setup** program will have created a program group and a program item, which is represented by the SCIL_Image icon:

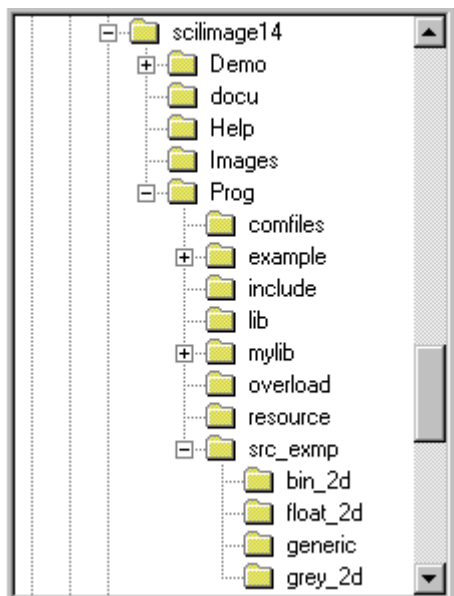


- 3) Before you can run SCIL_Image you will need to put the license code from the License Registration Form into the file **license.scl** in the **c:\scilimage14** directory. Use a standard editor to type in the license code, carefully copying it, including the difference between upper and lower case symbols.
- 4) Non demo licenses: Plug the hardware protection key (the *dongle*) into the printer port (the outlet port connecting the computer to the printer). Please note that you must have chosen the dongle-driver item during the installation.
- 5) On Windows95 some settings were added to your **autoexec.bat** file and on Windows NT some settings were added to the **system** control panel (in the Environment Tab).

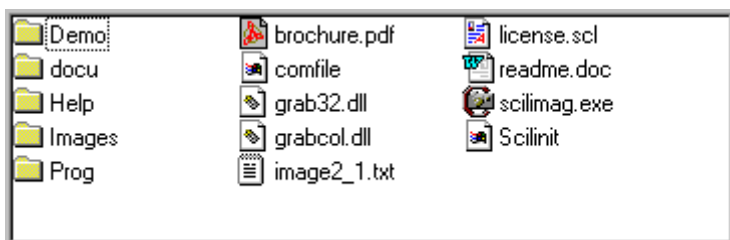
You could now start with chapter 3 'Getting Started', to get acquainted with the workings of SCIL_Image. Before you do, glance over the rest of this Chapter. It discusses the various folders and files, and also tells you what is involved in moving them elsewhere (section 2.3). Also, if you are not quite happy with the number and size of image windows SCIL_Image displays after starting up, you can change them by editing the file 'scilinit' (section 2.3).

The SCIL_Image Folder

In this section, we describe the contents of the SCIL_Image files and directories². The 'scilimag' directory contains a number of sub-directories:



In the main directory 'scilimag', you find the following:



brochure.pdf	a brochure about SCIL_Image
comfile	SCIL_Image command description file
grab32.dll	sample framegrabber dll
grabcol.dll	sample framegrabber dll
image2_1.txt	ASCII tekst file with additional notes
license.scl	file which must contain your license code
readme.doc	release notes regarding this version of SCIL_Image
scilimag.exe	the SCIL_Image executable

² The hierarchy of the files and directories in SCIL_Image was carefully planned, so please do not move files (or directories) to other directories before you know exactly what the files (directories) are used for. If you do move them, be sure that SCIL_Image can locate them (see Section 2.3).

scilinit the initialization file for SCIL_Image

It contains the following directories:

The 'demo' directory

This folder contains demonstration files with examples.

The 'help' directory

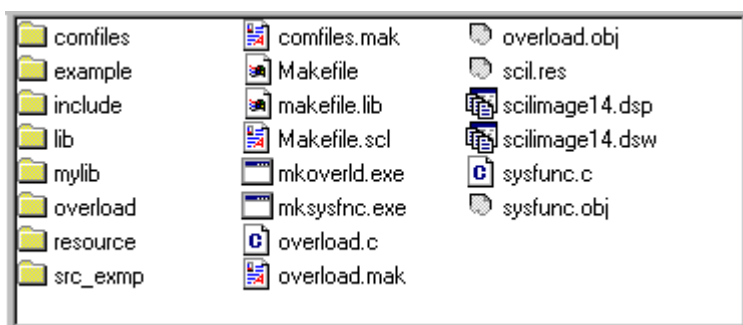
This folder contains SCIL_Image's on-line Help facility.

The 'images' directory

This folder contains images. Most of these images are used by the demos, so do *not* remove them in order to save space.

The 'prog' directory

The 'prog' directory is for SCIL_Image support, and contains files and programs which you need to build your own version of SCIL_Image. We will explain the procedure for that in Chapter 5.

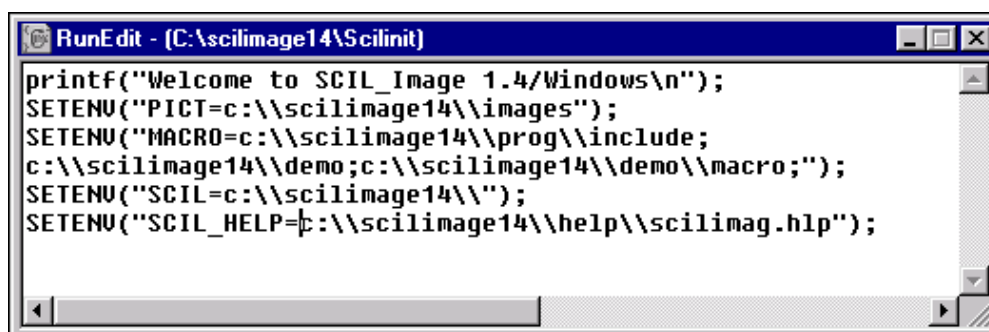


comfiles	directory with Command Description Files
example	directory containing some of the examples used in this manual
icons	directory with the icons used by the SCIL_Image user interface
include	directory containing include files
lib	directory containing the SCIL_Image functions
mylib	directory to store your own SCIL_Image functions
overload	directory with files for command overloading
resource	directory with resource files and icons
src_exmp	directory with sample C files referred to in Chapter 8
comfiles.mak	external makefile for Microsoft Visual C++
makefile.lib (.scl)	sub-makefiles used by 'makefile'
mkoverld.exe	program to create the file 'overload.c'
mksysfnc.exe	program to create sysfunc.c from the comfile
overload.c	c-file needed to create a new SCIL_Image executable
overload.mak	external makefile for Microsoft Visual C++

overload.obj	compiled overload file
scil.res	compiled resources
scilimage14.dsp	project file for Microsoft Visual C++ 5.0
scilimage14.dsw	project workspace file for Microsoft Visual C++ 5.0
sysfunc.c	c-file needed to create a new SCIL_Image executable
sysfunc.obj	compiled sysfunc file

Your SCIL_Image Environment

SCIL_Image resides on your computers in directories of your choice. You have to tell SCIL_Image what these directories are. When you have just installed the system, everything is where it is expected, and you can immediately start with the sample sessions in Chapter 3. Later on, you may want to move everything somewhere else. You then have to edit the file 'scilinit', in the c:\scilimage14' directory. You can do this from SCIL_Image, using the **File:Open C Program'** menu entry, which contains a standard editor.



```
RunEdit - (C:\scilimage14\Scilinit)
printf("Welcome to SCIL_Image 1.4/Windows\n");
SETENU("PICT=c:\\scilimage14\\images");
SETENU("MACRO=c:\\scilimage14\\prog\\include;
c:\\scilimage14\\demo;c:\\scilimage14\\demo\\macro;");
SETENU("SCIL=c:\\scilimage14\\");
SETENU("SCIL_HELP=\\scilimage14\\help\\scilimag.hlp");
```

There are four variables that tell SCIL_Image where its directories are on your computer. You have to use these variables to tell SCIL_Image where to find your data and programs. These variables are called *environment variables*:

- PICT** The folders where images are kept. At delivery this is set relative to the directory in which you chose to install SCIL_Image, so in our example: 'c:\scilimage14\images'.
- MACRO** The folders which contain programs, macros and include files for
At delivery set to: 'c:\scilimage14\prog\include, c:\scilimage14\demo, c:\scilimage14\demo\macro'.
- SCIL** The location of 'comfile'. At delivery set to 'c:\scilimage14'.

SCIL_HELP The file containing the on-line Help facility.

When you start to process your own images with SCIL_Image, you can either store those in the directory 'c:\scilimage14\images', or in a directory of your own. In the latter case, you have to change the environment variable **PICT** by adding on the names of the directories which contain your images.

When you write programs or generate macros, you can keep those in one of the directories listed in the '**scilinit**' file or in one of your own directories, which you must add to the environment variable **MACRO**.

SCIL_Image finds other initialization information in the '**scilinit**' file, for instance on the number of image windows it pops up when starting. Such information can be changed to your liking, except that you cannot ask for more than four (4) default images. Also, you should *not* change the order of the initialization statements.

The specific settings of the size and position of your SCIL_Image window, and of its subwindows such as the Worksheet and the History windows, are stored in a file '**scil.ini**' which SCIL_Image creates in the Windows directory. It is updated at the end of every session with your most recent preferred settings. When you restart SCIL_Image, it will look the same as when you last left it.

Chapter 3 Getting started

This chapter contains four sample sessions to get you acquainted with SCIL_Image.

Read this chapter if:

- You have never used SCIL_Image.
- You want to follow some typical action scenarios.
- You want an overview of the menu contents.
- You want to know how different interactive tools can be combined.

Do not read this chapter if:

- You are familiar with the SCIL_Image menu system and command interpreter.
- SCIL_Image has not been properly installed (see Chapter 2).

Before you begin

You should already be familiar with basic Windows concepts such as:

- Using icons
- Using the mouse actions - *point*, *drag*, *click* and *double-click*
- Pulling down menus and choosing commands
- Scrolling in windows
- Resizing and moving windows
- Terms such as *dialog box*, *list box* and *button*

If you are not familiar with these, consult your Windows manual.¹

The Five Modes of Interaction with SCIL_Image

There are five ways in which you can engage in image processing with SCIL_Image. They are, in order of complexity:

- viewing images and using interactive tools
- using mouse and menu
- command line typing and use of 'macros'
- making your own interpreted programs and reaching them from menus
- making your own compiled version of SCIL_Image

¹ SCIL_Image was originally written for systems using the UNIX operating system. As a consequence, it has a few properties that are not quite 'standard Windows behavior'. For instance, image processing commands require the name of the image explicitly as an input parameter - they do not just work on any image you select with the mouse, as you might expect. We will point out those non-standard properties at the appropriate locations.

We will explore these modes of interaction in five sample sessions. These sample sessions are not exhaustive, but they give you an impression of what can typically be done in each of the interaction modes. We end this chapter with an overview of the SCIL_Image commands that can be found in the menus. A lot of the factual information of this chapter may be found in the Help pages of SCIL_Image (in SCIL_Image, press **Ctrl-H**, go to the 'Contents' menu of the Help facility, and press '>>' or select a topic of your choice).

Session One: Viewing Images

To start SCIL_Image:

double-click on the SCIL Main Window icon



Figure 3-1 : The SCIL_Image program icon

If all is well, the system displays a menu bar, four windows named A, B, C, and D at the top of the screen, two text windows named Worksheet and History, and a status bar with tool buttons at the bottom. (If you are using a small screen display some of these windows may not be fully visible. Also the image windows will be black at start-up - in the picture below, we have put in an image for added interest.)

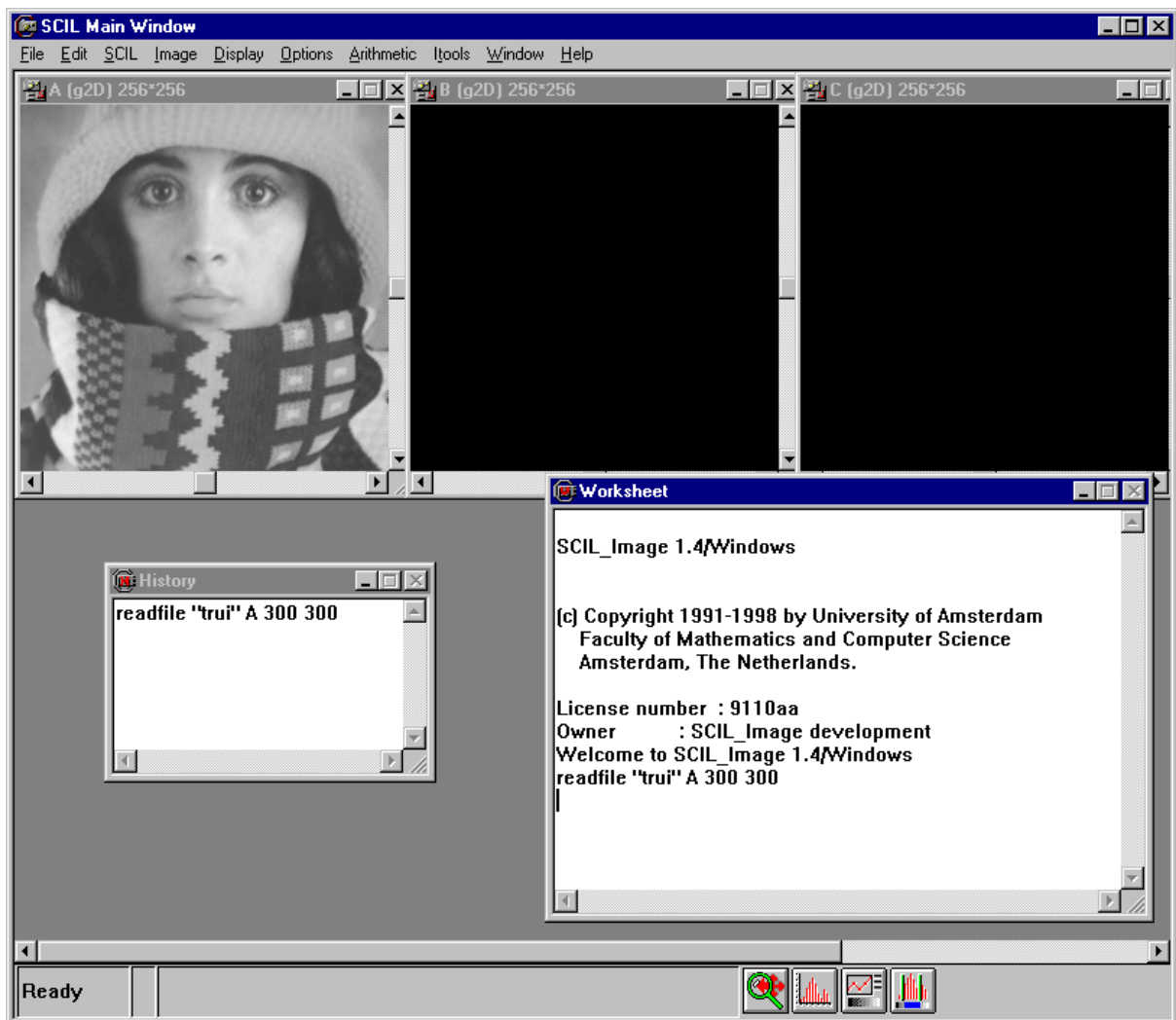


Figure 3-2 : The opening screen of SCIL_Image

The windows you see are all standard windows, which may be *moved around*, *resized* and *scrolled* in the standard Windows way, as well as *minimized*. The bar at the bottom of the SCIL Main Window has a status indicator on the left which should now read 'Ready'.

The system is ready to receive its first instruction. In this session we concentrate on display of images, so first, open an image.

Point with the mouse at the 'Image' menu; click and hold. A menu will appear. Move the mouse pointer, while holding down the button, to the item named 'I_O'. Another menu appears; move the pointer to 'readfile' and release.

(You can actually also do this by clicking on the menus that appear, rather than holding the mouse.) From now on we will abbreviate mouse action sequences. Thus the above sequence becomes:

select 'Image:I_O:readfile'

The menu will disappear and a *SCIL_Image dialog box* will pop up.

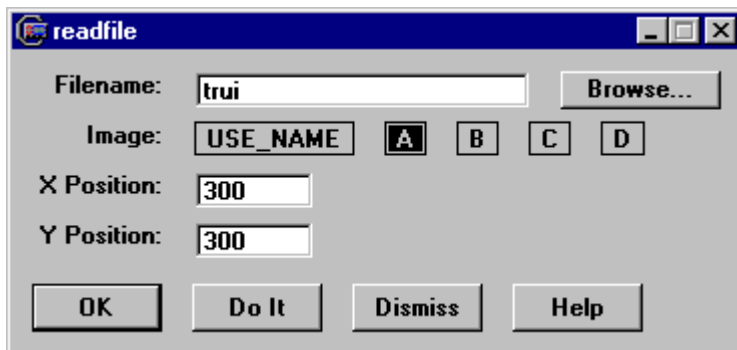






Figure 3-3 : A dialog box

A *SCIL_Image* dialog box may contain several items. The command you just selected is the title of the box - in this case, **readfile**. The four buttons at the bottom are found in all dialog boxes.

-  button executes the command and removes the dialog box.
-  button executes the command but keeps the dialog box.
-  button gives on-line help information about the command .
-  button closes the dialog box (it has the same effect as clicking the close box in the upper left corner of the dialog box).

(There is no need to make the dialog box disappear before you open up another one - *SCIL_Image* will retain the last one if they conflict). The default image 'trui' is not very interesting (you may view it by clicking 'OK'.) To select an image from the library that is included with *SCIL_Image*, select the browse button.

click 'Browse'

A dialog box 'Open' appears, in which you can select an image file in the normal way you select files in Windows. The images are in the sub-directory 'images' of the directory 'scilimag'. You will find a number of files. Let us select 'schema.dat', by double clicking.

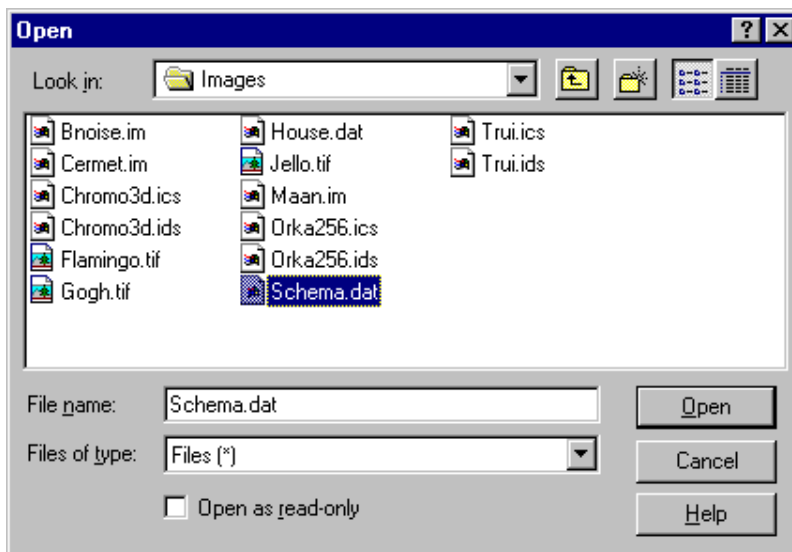


Figure 3-4 : The file browser.

The Open dialog box disappears and we return to the 'readfile' dialog box.

click 'OK' or 'Do It'

The command 'readfile' is now executed and an image appears in window A. Note that at the same time, the text:

readfile "c:/scilimage14/images/schema.dat" A 300 300

appears in both the Worksheet and the History windows (the latter may still be an icon). This will always happen when you execute a command; we will demonstrate the use of these windows in Session Two.

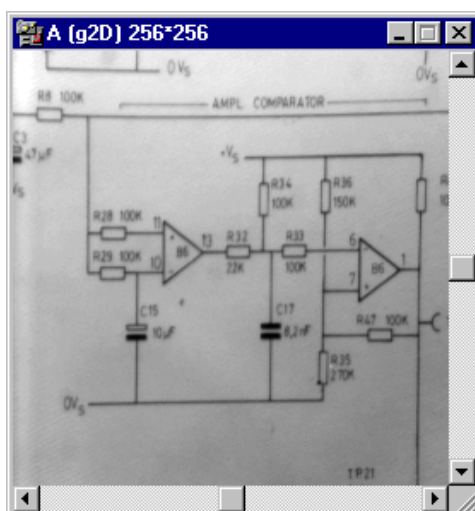


Figure 3-5: The "schema" image

The image is displayed in the window labeled A. This window is called a *viewport* to image A; you may define several viewports for the same image (which we will do later in this session). When you use the scrollbars of the viewport, you see that they move the viewport over the image.

scroll window A using the scrollbars

This is useful when looking at parts of large images. On the "schema" viewport, a black border scrolls in, to indicate where the actual image data ends. You can scroll only until one of the corners of the image reaches the center of the viewport.

The image in A is a two-dimensional, grey-valued image, of 256 by 256 pixels (picture elements). This is indicated on the title bar by `'A[g2D]256*256'`. You can inspect the individual pixels by pointing and clicking:

click on image A to select it

click again on image A and drag

In the status bar, a set of numbers appears which change when you drag:

[156,112] 214

The first two numbers indicate the position where you clicked, the third number is the grey-value. For an image of type `'g2D'`, this is an integer. The position numbers are the coordinates of the *pixel* (picture element) of the stored image, not of the point in the viewport. (For example, the capacitor on the right always has coordinates [150,164], no matter how the viewport has been scrolled.). There are a number of image types in SCIL_Image, including real-valued images [f], complex-valued images [c], binary images [b] color images [col]. We will come across them in the course of the next demo session.

You can have several viewports connected to the same image. Let us generate a second viewport on image A:

select 'Display:create_display'

select 'Image to connect display to' : A

click 'OK'

A new viewport appears, also labeled `'A[g2D]256*256'`, since it displays the same image. To show that these viewports are indeed displaying the same image, let us perform an operation on the image:

select 'Image:Histogram:equalize'

select 'Output Image' : A
click 'OK'

Image A is processed in-place, and the changing result is immediately reflected in both viewports. The use of several viewports on the same image can be combined with the tool buttons on the status bar. You select *viewports* with the *right* mouse button (whereas you select *windows* with the *left* button):

select one of the viewports on A by clicking on it with the *right* mouse button

A green border in the viewport indicates that it has been selected.

click the zoom-and-pan icon



A 'Zoom & Pan' interaction window pops up, with a miniature viewport displaying image A.

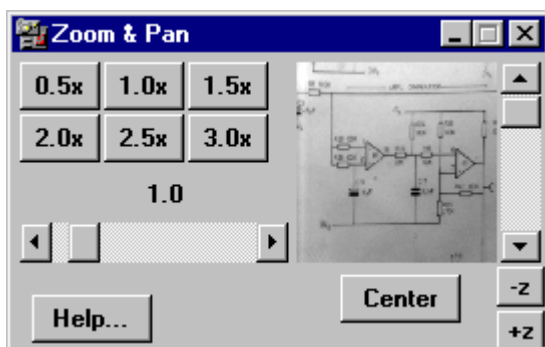


Figure 3-6 : The Zoom & Pan window

When you point and drag on the little copy of A in the 'Zoom & Pan' window, the selected viewport changes as if you were scrolling it. When you click on one of the numbered buttons, the zoom factor of the display changes - you may also change this more continuously with the sliding bar.

play around with the 'Zoom & Pan' options

Again, a change in image A is reflected in all viewports. You may verify this by reading in the original image:

select 'Image:I/O:readfile'
click 'Browse'
double-click 'c:/scilimage14/images/schema.dat'
click 'OK'

All three viewports on image A change their displayed contents.

Zoom & pan can handle several images at the same time. To show how this might be useful, we do some simple processing on image A: we *threshold* the image, that is we make a binary (two-valued) image in which everything lighter than a given grey-value is made *object* (displayed as red) and everything darker than that grey-value is made *background* (displayed as black).

select 'Image:Conversion:threshold'

click 'OK'

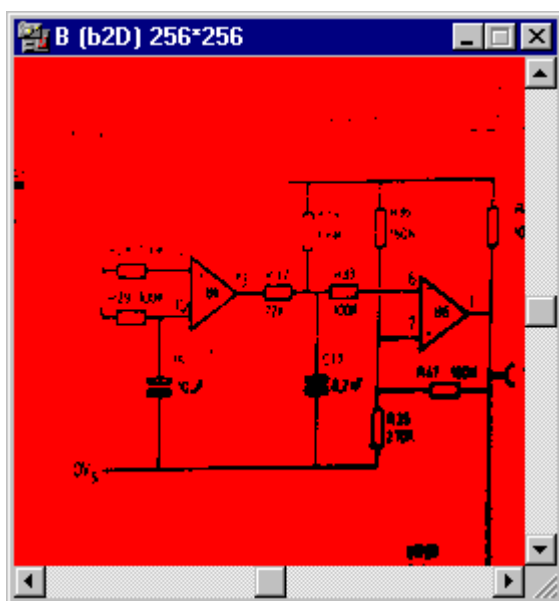


Figure 3-7 : A thresholded image

A binary image now appears in window B, marking all pixels that were less than 128 in greyvalue red (you may check this by the point-and-drag method to display positions and values in the status bar.) We can use the zoom-and-pan tool to examine this in more detail:

select image B with the right mouse button

A green border appears to show it has been selected. Note that there is also still a green border around the selected viewport on image A. Now when you zoom-and-pan, the display in both viewports is affected.

play around with zoom-and-pan

When you de-select the viewport A from the zoom-and-pan tool (by a right-mouse click in the viewport), viewport B is then displayed in the interaction window, and the actions only affect that viewport. On the de-selected window, the final display settings of zoom and pan

remain. Thus you can use the zoom-and-pan tool to permanently adjust the display of your images. You can close the zoom-and-pan tool completely by double-clicking on its close box, or by minimizing it. Again, the final display settings then remain valid on the selected images. You might as well destroy the extra viewport on image A, since we will not be needing it any more:

double-click on the close box of the extra viewport A
click 'Viewport only' in the dialog box

Zoom-and-pan can also be used to view color images, or 3-dimensional images.

select 'Image:I/O:readfile'
click 'Browse'
double-click 'c:/scilimage14/images/chromo3d.ids'
click 'OK'

This is a 3-dimensional image of a cell division. When you select it with the right-mouse button and activate zoom-and-pan, you will see that its dialog box contains buttons for +z, -z and a scroll bar. These can be used to step through the stack of 2-dimensional images that form the 3-dimensional image.

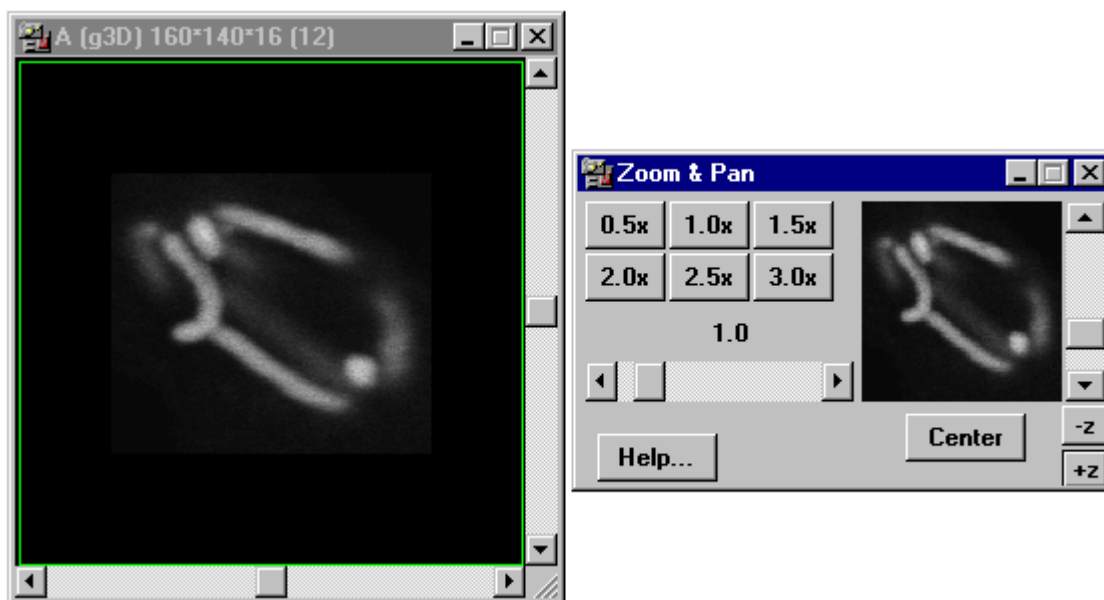


Figure 3-8: Viewing Z-slices

A utility that differs from zoom-and-pan in a useful way is **lens**. The **lens** command allows you to construct an *image* that contains only part of image A, instead of just a displayed viewport. Let us read in a grey-valued image:

select 'Image:I/O:readfile'
click 'OK'
select 'ITools:lens'
select 'Lens image': B

Image B becomes a 31*31 image (since those are the default values for lens). To view it properly, select it with the right mouse button, click on zoom-and-pan, make the zoom factor 8, then close the zoom-and-pan. Now you can move your mouse through image A, and see the data in image B change along with it.

The other buttons on the status bar contain interactive tools that can be used for image display, and also for simple image processing. They deal with grey values, and are therefore not applicable to all image types. The interaction tools can also be used in combination. For instance, we can have the histogram of image A even when we are using the 'lens':

select image A by a left-mouse click
click the Histogram icon



You can grab the green bar to read grey value and frequency from the histogram. We can even have a histogram of the lens image B at the same time, updated while we are moving the lens:

select image B by left-mouse click
click the Histogram button
move around in image A by dragging the lens

The histogram changes as the image in B changes because the lens position changes. When you are done with the lens, you have to switch it off explicitly, using the menu:

select 'ITools:stop_lens'

Another histogram-based utility is the Threshold Editor, which allows you to select an appropriate threshold for an image by looking at its histogram.

select image A
click the Threshold Edit icon



The histogram of A is displayed, and by dragging the green bars you can see the effect of a thresholding on A (you can move both bars at the same time by dragging in the middle).

experiment with Threshold Edit

When you are satisfied, you may exit by clicking 'OK' . This executes the thresholding in image A with the settings chosen; a binary image is the result.¹ If you cannot find a good threshold, and therefore would not like to do the thresholding operation, you should exit the tool by clicking 'Cancel', which returns A to its original grey values.

The final interactive tool is the Grey Map Editor.

select image A

click the Grey Map Editor icon



The Grey Map Editor interaction window displays a function which maps grey values of image A to other grey values. You can change this function by grabbing the nodes of the graph and dragging them (to add or delete nodes, click the right mouse button). All displayed copies of image A change accordingly. When exiting the Grey Map Editor, you have the choice to keep the changes to the image (OK), or to restore the original grey map (Cancel).

Note through all of these actions that SCIL_Image allows you to have multiple interaction windows open, and does not require you to close one before you can interact with the next. Its way of interaction is 'modeless'.

This concludes the first sample session. Should you want to stop now:

select 'File:Exit SCIL'

¹If you still had the histogram of A displayed, you will notice that it becomes void, and displays the error message: IMAGE TYPE DOES NOT SUPPORT HISTOGRAMS.

Session Two: Using the SCIL_Image Menu System

The following sample session introduces various aspects of image processing using SCIL_Image through its menu and dialog system. Not all topics are covered, but you should get an impression of how to work with the system.

Let us read in an image.

select 'Image:I/O:readfile'
click 'OK'

This is a 256*256 grey value image, as its label 'A[g2D]256*256' indicates. The grey values are integers (in this case ranging between 0 and 255). You can point-click-and-drag at the image and obtain a display of the grey values in the status bar at the lower left.

Let us inspect some of the *grey-value filtering* operations in SCIL_Image. A *laplacian* determines a local (and rather coarse) approximation to the second derivative of the image:

select 'Image:Filter:laplace'
click 'OK'

When you look at the values of this image, you will see that they may be negative. They are displayed clipped to 0 and 255, so -126 is just as dark as 0. This is the standard lookup table for the display of g2D-type images. You can check the pixel values by dragging the mouse in the image and reading the values on the status bar.

There are quite a number of image processing operations valid for image type g2D, both the standard linear filters (convolutions) and some non-linear filters such as percentile filters and the edge-enhancing Kuwahara filter:

select 'Image:Filter:Kuwahara'

Parameters are pre-set to do the operation from 'A' to 'B'. Before executing the filter, change the filter size from '5' to '9' to make the effect more pronounced.

choose Filter Size '9'
click 'OK'

You will have to wait a little while since this is quite a complex operation. During the operation, the status indicator in the lower left corner changes to 'Running', and an hourglass icon appears next to it.

This may be a good time to experiment with some of the filters in 'Image:Filter', to acquaint yourself with the standard capabilities of SCIL_Image. If you want a description of a command, just press 'Help' in its dialog box, and a Help page will pop up. For **kuwahara**, this reads:

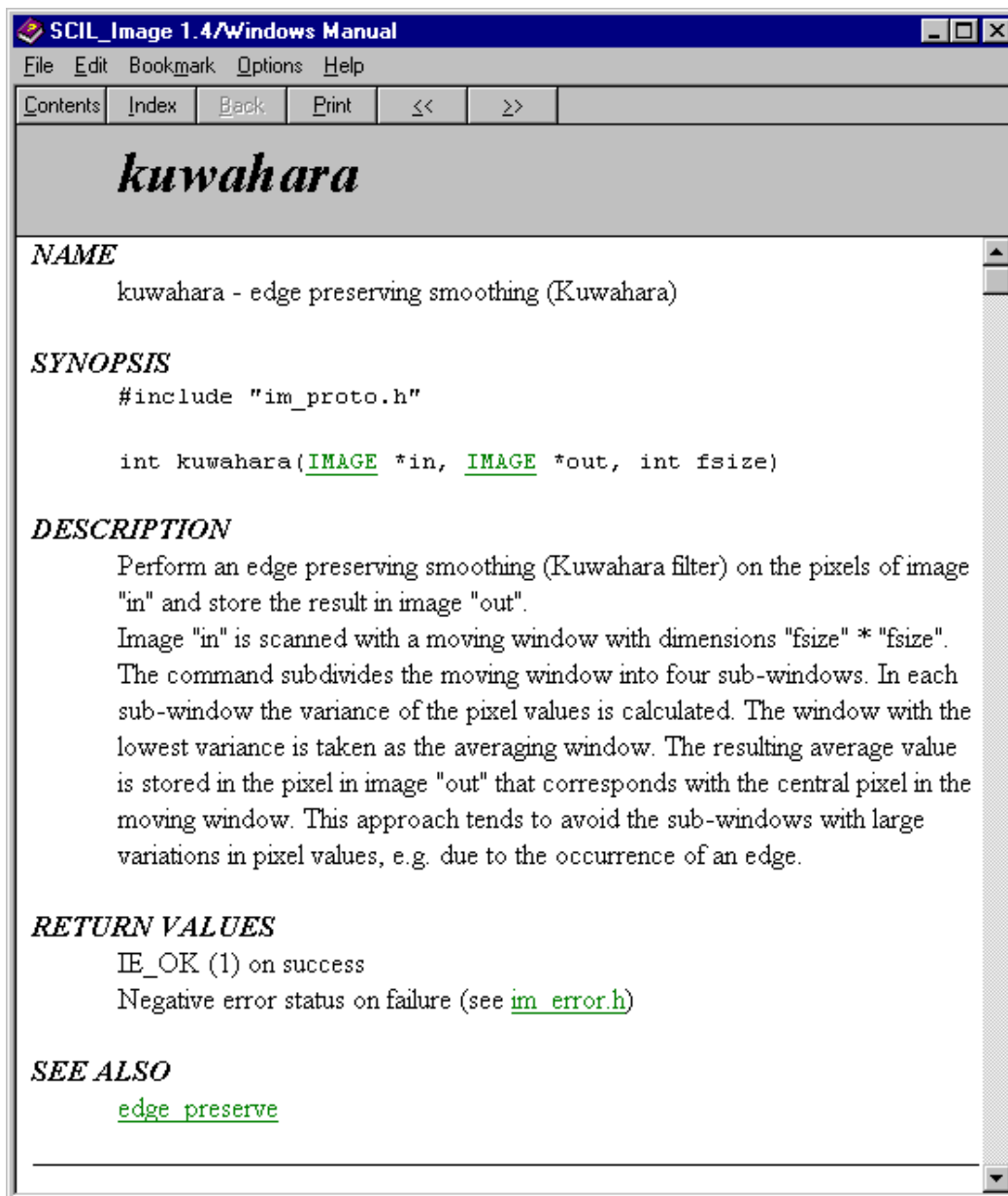


Figure 3-9 : A reference manual page

You can browse through this Help facility in the usual way, using 'Contents', 'Search', links (which are underlined) or the '>>>' and '<<<' keys, if enabled.

For the next illustration of SCIL_Image, we use the 'fast-fourier transform', which computes the frequencies in an image. Even though this transform uses a fast algorithm, it still takes quite a while on a standard size image. So, we reduce the image first:

```
select 'Image:Manipulation:change_image_size'  
choose Image 'B'  
choose New Width '64'  
choose New Height '64'  
click 'OK'  
select 'Image:Manipulation:warp_image'  
choose Input Image 'A'  
choose Output Image 'B'  
click 'OK'
```

Image A was an integer-valued image consisting of 256*256 pixels (its header is "A(g2D)256*256"), image B consists of only 64*64 pixels (its header is "B(g2D)64*64"). The 'warp_image' command performed the conversion. Note that the much reduced image B looks small in its viewport. We can display it much bigger by using the zoom-and-pan button at the bottom of the screen. We explained how to do this in the previous session:

```
select image B by a right-mouse button click  
click on the zoom-and-pan icon  
use the slide bar in the dialog box to set the zoom size to 4.0
```

Now we do the fourier transform on the reduced image:

```
select 'Image:Transform:fast_fourier'  
choose Input Image 'B'  
choose Output Image 'C'  
choose 'forward'  
click 'OK'
```

Image C also appears small in its viewport. You would like to have it zoomed in by a factor of 4.0, just like B. You can do this by a single action:

```
click on viewport C with the right mouse button
```

The zoomer instantly applies to C as well (note the green border around C).



Figure 3-10 : The Fast Fourier Transform

The header of window C reads "C(c2D)64*64". This 'c2D' indicates that the window contains a 2-dimensional image of which the value at every pixel is a complex number. You may check this by pointing at it - the status bar at the bottom of the screen indicates the complex values.

SCIL_Image discriminates carefully among the various types of images. Not all operations are implemented for all image types. For example, if you try to do a Gaussian filter on C

- select 'Filter:gauss'**
- choose Input Image 'C'**
- choose Output Image 'D'**
- click 'OK'**

The command does not execute, and you get a warning box:

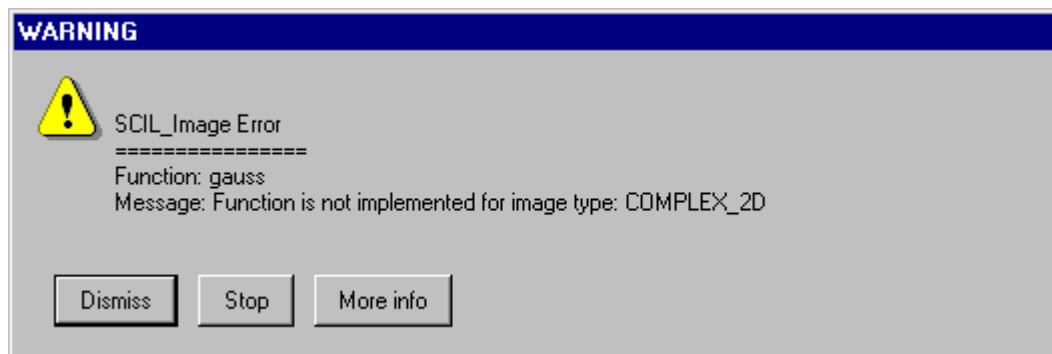


Figure 3-11: An alert box

You exit from this by clicking 'Continue'?

You can convert the complex image to a *float* (real-valued) image by taking its real part, using the command found 'Arithmetic:Complex_based' :

```
select 'Arithmetic:Complex_based:real_im'  
choose Input Image 'C'  
choose Output Image 'D'  
click 'OK'
```

The type of the new image in window D is 'f2D', a floating point image. The image values are not integers between 0 and 256, as in the case of 'g2D', but real numbers (you may check this by the click-and-drag method of viewing the image content pixel by pixel, with display in the status bar). Some of the numbers are negative, and they are displayed as black.

Somewhere in the analysis of any image, you will probably create a *binary image*. This is a two-valued image, usually used to denote which pixels belong to objects, and which to the background. Let us take an example where we want to perform measurements on the objects in the image:

```
select 'Image:I_O:readfile'  
click the 'Browse' button  
find the file 'scilimage14/images/cermet.im'  
double-click on this file's name  
click 'OK'
```

To convert this into an appropriate binary image, do the following:

```
select 'Image:Conversion:isodata_threshold'  
click 'OK'  
select 'Arithmetic:invert_im'  
click 'OK'
```

This produces a binary image (note the type: b2D). Let us first demonstrate some filters on binary images from mathematical morphology:

```
select 'Image:Morphology:erosion3x3'
```

² When you give commands one at a time, there is no difference between 'Dismiss' and 'Stop', but if you were running a program there is. 'Dismiss' skips the erroneous command but still executes the rest of the program, while 'Stop' halts the execution of the program completely.

choose Input Image 'B'
choose Output Image 'B'
click 'Do It'

The image in window B has changed slightly. A more pronounced effect is achieved by repeating the command a few times. This is the reason we used 'Do It' rather than 'OK': it leaves the dialog box visible.

click 'Do It' 4 more times

Restoring the rough outline of the shape is achieved by using the dilation operation 5 times:

select 'Image:Morphology:dilation3x3'
choose Number of iterations '5'
click 'OK'

This should give you a sense of the speed of the implementation. By the way, while we are in the morphology menu, a transform in SCIL_Image related to morphology is the distance transform:

select 'Image:Morphology:distance'
choose Input Image 'B'
choose Output Image 'C'
click 'OK'

A *distance image* will appear in window C. Every point in this image has a grey-value proportional to the distance of that point to the closest object. (You could also do the erosions by thresholding the distance transform image.)

The binary image in window B contains many objects. SCIL_Image can perform measurements on these objects:

select 'Image:Single_Objects:measure'
choose Interaction 'YES'
click 'OK'



Figure 3-12 : The Imeasure dialog box

A new window called 'labelled_image' appears, with the objects in various colors.

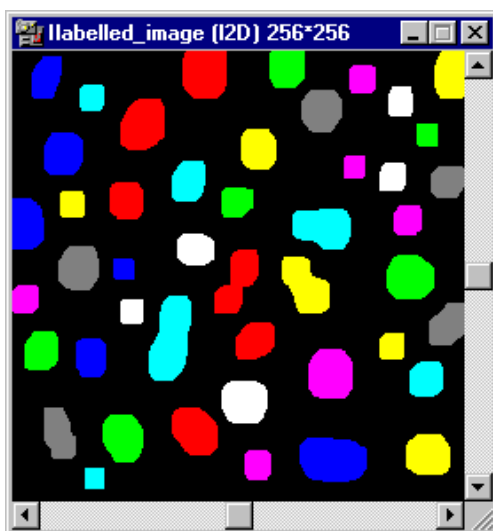


Figure 3-13 : A labeled image

click at individual objects in this window

In the Worksheet window, a list of feature values grows, reporting on the shape measurements you selected. To conclude measuring:

point at the 'labelled_image' window

hit the <Enter>-key

This is the only way you should conclude the measuring³. You could have written the measurements to a file using the 'Store in file' option of the 'measure' dialog box before starting the measurements (use the 'Browse' button next to the bar to look through your directories).

You may need more image windows than the standard windows A, B, C and D, for instance to store interesting results. You can make a new image window by:

```
select 'Image:make_image'  
type as Image name 'my_image'
```

The image can have any name (as long as it does not contain a white space), type, size and location you want, by selecting the appropriate options in the dialog box. Creating small sub-images may be helpful in selecting details of larger images, in which case you may find the 'copy_part_image' command from the 'Image' menu useful.

When you select a command, 'Image:copy_im' for instance, the choice of images includes your newly made image 'my_image' *You should realize that the image is lost when you exit SCIL_Image, unless you save it using 'Image:I_O:writefile' . SCIL_Image will not prompt you to save the images, you have to do it yourself.*

This ends the second sample session. To quit:

```
select 'File:Exit SCIL'
```

³ Clicking in the close box of the labeled image will destroy that image, but not terminate the running program. In that case, you would still have to stop it, by using the Pause/Break key, and Enter. Your data is then not written to a file.

Session Three: Using the SCIL_Image Command Line Mode

This sample session introduces the various aspects of *command line operations* in SCIL_Image. It teaches you how to give commands by typing rather than by using the menu. You will need this ability to write programs in SCIL_Image. Please work through this session by keying in the examples. We assume that you have completed the previous sample sessions. Restart SCIL_Image if it is not still active:

double-click the 'SCIL_Image' icon

In the previous session, we showed how SCIL_Image can be operated using the *menu* system. In this session, we will use SCIL_Image as a *C-interpreter* (C being the programming language). This means that if you type statements with correct C syntax, the system will execute them without the usual sequence of compiling and loading normally required when programming in C.

For instance, to have the system print some text, you can type this C-statement (copy this line precisely, including the semicolon) in the window 'Worksheet':

printf("Hello, world\n");

hit the 'Enter' key on the lower right of the numeric keypad

The 'printf' line is a legal C-statement which will be executed immediately after you hit the 'Enter' key. The result is that the Worksheet shows:

Hello, world

(We will use ***bold italics*** to indicate program output.) Notice the difference between the use of the two 'Enter' keys on your keyboard in all text windows of SCIL_Image:

- | | |
|---|--------------------------------------|
| 'Enter' on the keyboard (the Return key) | - gives a new line, does not execute |
| 'Shift-Enter' on the keyboard | - executes a line or selection |
| 'Enter' on the numeric keypad | - executes a line or selection |
| 'Ctrl-Enter' (either of them) | - pops up dialog box for a selection |

(We prefer the use of the Enter on the numeric keypad.) Now type the command:

readfile trui A <hit numeric keypad 'Enter'>

This instruction is *not* a legal C statement, therefore it can not be executed directly. However, it is first passed through SCIL_Image's *scommand expander*, which translates such typed-in instructions into legal C statements (if possible). The instruction is then executed in the same way the **printf** statement was. As the result of the above command, an image is read from disk and displayed in image window A. If translation into legal C is not possible, the system issues an error message. For example, try mis-spelling the command:

```
readfile trui A <hit 'Enter'>
```

The system responds with:

```
readfile t --> variable used but not declared
```

The same command 'readfile trui A' can be issued through the menu system, as described in the previous sample session. Do so now:

```
select 'Image:I_O:readfile'  
click 'OK'
```

Note that the command appears in the Worksheet as **readfile "trui" A 300 300**. You can use both types of interaction: command-line typing and menu selection, intermixed. Of course, you may have forgotten what the options are - the menu system always prompts you, but the command-line system does not. In that case, you can just type a '?' after the command, and it will ask you for all the parameters that it needs.

```
type 'readf ?'
```

The system prompts you for the arguments of the **readfile** command, one by one. You can either press the return key when you like the default value, or enter your desired value. You can also pop up the dialog box of a command, by selecting the command in the Worksheet and hitting 'Ctrl-Enter' (either of the Enter keys will work):

```
type 'readf', select it by double-clicking on it, and hit 'Ctrl-Enter'
```

An important feature of SCIL_Image is that C control statements can be mixed with image processing commands, such as in:

```
int i; /* declaration of a variable */  
for(i=0; i<200; i = i + 10 ) thresh A B i;
```

As you see, this creates a for-loop which thresholds the image at different levels. The variable 'i' is now known to the system, so if you want to see this again, executing only the second line

is enough. You need not re-type this line, just execute it again by selecting it (and then clicking on 'Enter').

You can *interrupt* the execution of any command by using the **Pause/Break** key. Run the previous command again, and interrupt it - SCIL_Image is then ready for the next command.

The Worksheet, in combination with the mouse, offers a flexible way of giving commands:

- If you want to *execute more than one command* in the Worksheet, you select them by using the click-and-drag method until all desired commands are highlighted; then hit the 'Enter' key.
- If you want to *re-execute a command*, you can do this simply by 1) moving the cursor to that line in the Worksheet, 2) clicking to activate the line (it will not look selected, but it is), 3) hitting 'Enter'.
- You can also *edit a command*, using the normal method (mouse selection and re-typing), before you re-execute it. In the same way, you can remove lines that you do not want.

Another way to re-execute commands is provided by the 'History' window. In that window, all commands you have given are saved in the order you gave them. You can reach the History window under the 'Window' menu:

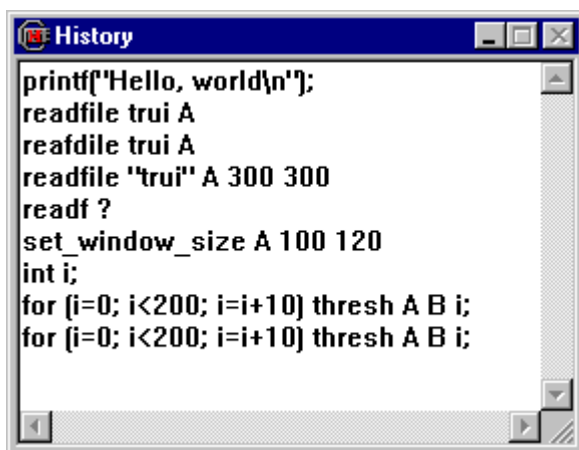


Figure 3-14 : The history window

When you re-executed the **for**-loop, it was listed in the history window a second time, even though you did it by selecting the line in the Worksheet that was already there. The History window is a faithful account of what you did. It can *not* be edited. You select lines in the same way as for the Worksheet and execute by hitting 'Enter'.

SCIL_Image can pop up three different kinds of text windows. These are the Worksheet, the History window, and the RunEdit window (which you create when you choose 'File:New C

Program’, see below). In any of these windows, you can select text with the mouse - and if you then hit the ‘Enter’ key, SCIL_Image will attempt to execute the selection.

If you perform, in any text window, the actions:

- select a command name by mouse**
- hit ‘Ctrl-Enter’**
- in the dialog box which pops up, select ‘Help’**

then SCIL_Image will pop up a Help page with information on the command.

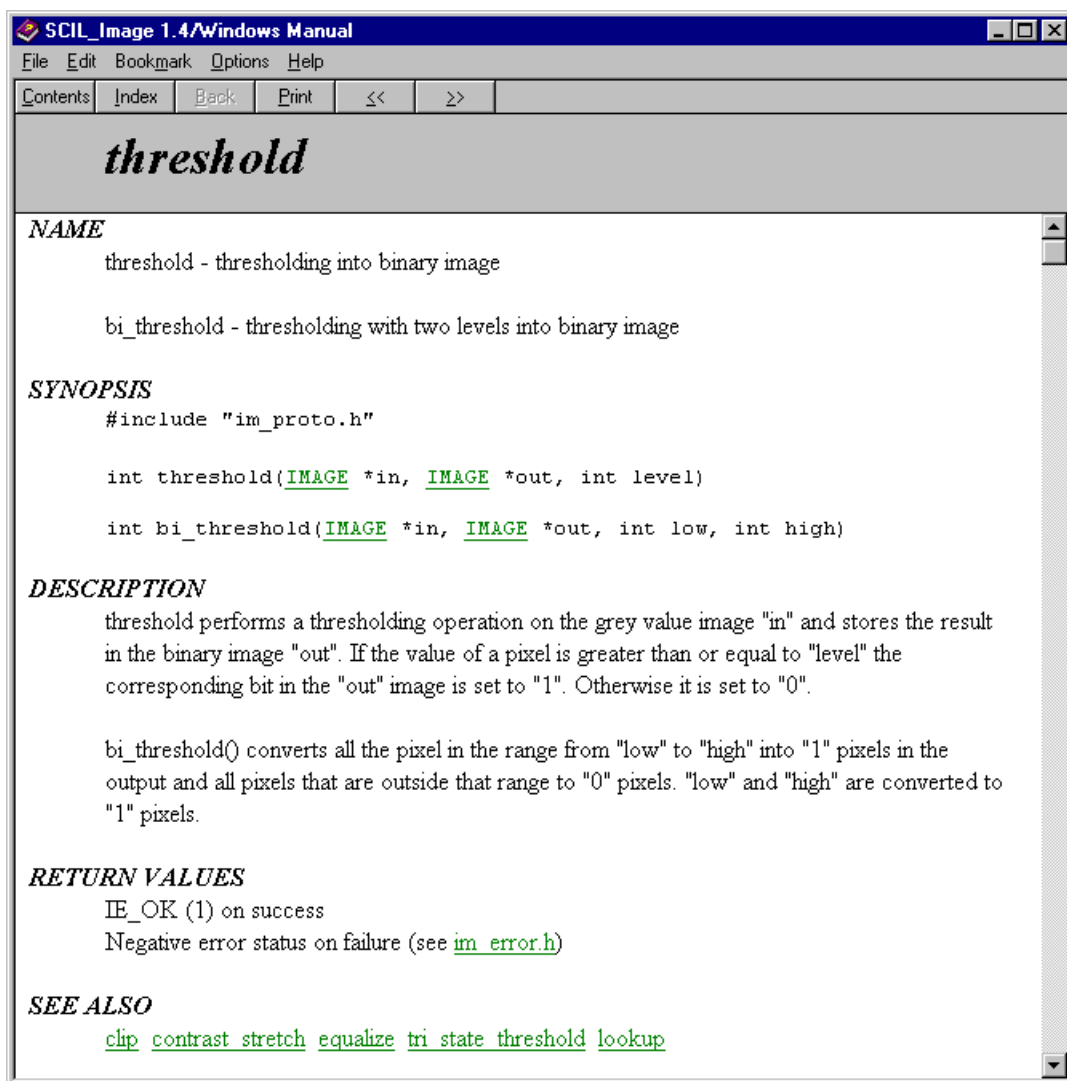


Figure 3-15: Another manual page

An alternative way to find this Help page is:

- hit ‘Ctrl-H’**
- click the Search button**

type the command name

Proceeding with the example, the last command:

```
for(i=0; i<200; i = i + 10 ) thresh A B i;
```

is equivalent to:

```
for(i=0; i<200; i = i + 10 ) threshold(A, B, i);
```

Edit the former to produce the latter, and execute it. The whole line is now a proper C-statement, which does not need to go through command expansion to be executed since it contains an explicit call to the C-function `threshold()`. Please be aware of the difference between C-statements and non-C-statements. If you want to know the proper C syntax for a command called `<command name>`, call up its Help page by the procedure described above.

Note carefully that although it is possible to mix legal C syntax with non-C commands in the SCIL interpreter, it is strongly recommended to *use only legal C syntax in your programs* because they could then be compiled (which will make them faster) and added to a library (so you can use them as functions in your programs).

Finally, we show you how to collect a series of commands in a file which can be executed. Such a file is called a *macro*. Let us make a macro for the set of commands that gave us the contour of the 'trui' image. Type in the commands:

```
readf trui A  
thresh A B  
contour B
```

Open a new file with 'File:New C Program' . This pops up a RunEdit window. Copy the last two commands from the Worksheet into the RunEdit window, using 'Edit:Copy' . Save the file as 'contourA.mac' in the directory 'scilimage14\demo\macro' . You can execute the commands in this macro by typing:

```
macro contourA.mac
```

The macro makes the contour of any grey-valued image in window A and displays the result in window B.

In this way you can collect useful command sequences. However, you cannot give parameters to macros, so if you would need a macro to make the contour of the image in B, you would have to write another macro. In that case, it is better to write a *program* (see the next section "Session Four: Programming in SCIL_Image"). Macros are very useful for writing demos

(which use the same images, and display results in the same way), or for saving effective sequences of commands from a session.

This concludes the second sample session. To quit SCIL_Image, type:

exit

Session Four: Programming in SCIL_Image

In this fourth session, we will build a small program. We show you how to incorporate this program as an *interpreted command* into SCIL_Image. We also demonstrate how to bring this program into the SCIL_Image menu system.

The program will read an image from a file, threshold the image at a given level and then show the contour in a display window. We call this program 'myfunc'. We would like to give it an file name, an image, and a threshold value as arguments. Make the new file 'myfunc.c' by selecting 'File:New C Program' from the menu. Type in the contents of the box below:

```
#include "image.h"

myfunc(filename, image, level)
char *filename;
IMAGE *image;
int level;
{
    readfile(filename, image, 0, 0);
    threshold(image, image, level);
    contour(image, image, 0, 8, 0);
}
```

Let us use this opportunity to show you some features of SCIL_Image that are very convenient when you are developing programs. Suppose that you had forgotten exactly what the arguments were of the function `readfile()`. You can easily pop up the required information *without leaving the editor*. You simply select the text 'readfile' with the mouse, and type '**Ctrl-Enter**'. The dialog box pops up, and by clicking the **Help** button you obtain the Help page on the readfile command. Note that in the dialog box, you can also see what the default values of the parameters to this function are.

Select 'File:Save' from the menu, and:

- give it the name of the main function in your program, followed by '.c' - in this case 'myfunc.c'. The name of your file should *always* be the name of the main function defined in it, followed by '.c'.

- store it in a directory contained in the `MACRO` environment variable (see Chapter 2), for instance in the `'scilimage14\prog\example'` directory.

If you do not do this, `SCIL_Image` cannot find your function.

You can *not* use this function using the `'File:Run'` command - that is reserved for complete C programs (which should contain a `main()`). To use the function, exit the editor by clicking on the close box of the `'myfunc.c'` window (upper left button). *Do not* use `'File:Exit SCIL'` to quit, it will quit `SCIL_Image` altogether!

You will automatically return to `SCIL_Image`. From `SCIL_Image` you can now use the function `myfunc()` by typing in its name and list of parameters:

```
myfunc("cermet", a, 127);
```

When the `'Enter'` key is hit, your function will be loaded and executed immediately. You can *not* abbreviate your function to a `SCIL_Image`-style command `myfunc cermet a 127`, because that could only be done if the function would have an associated `'CDF'` file (see chapter 4).

We will demonstrate below how to do that. In **“Making a New Compiled Version of `SCIL_Image`”** we show how you can incorporate your function into `SCIL_Image` as a full-fledged, compiled, command.

To help you manipulate your own programs in `SCIL_Image`, there are some commands you may find useful. They can be found under the `'SCIL'` menu, but may also be typed in. We discuss some of them now.

If, for example, you wish to remove the current program from `SCIL_Image`'s memory use the command:

```
rmvar
```

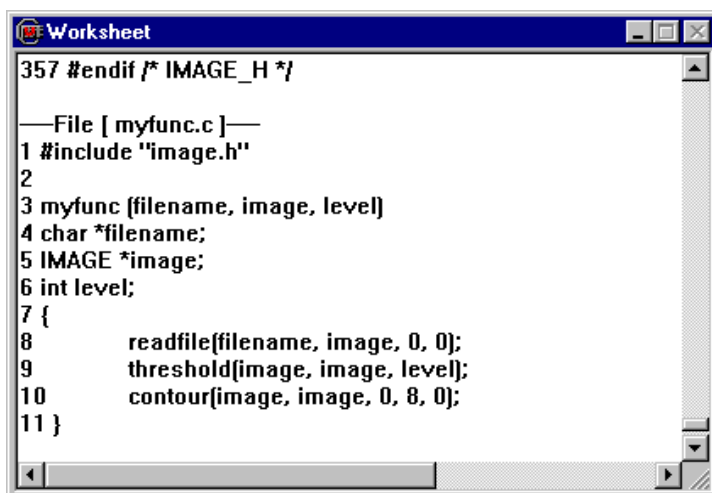
The program and all variables connected with it have now been deleted. You can reload the program by typing:

```
load myfunc.c
```

To inspect the text of the loaded program:

```
list
```

Several windows of text scroll by of the include file `'image.h'` (the first line in your program file) and the Worksheet ends up looking like this:



```

357 #endif /* IMAGE_H */

—File [ myfunc.c ]—
1 #include "image.h"
2
3 myfunc (filename, image, level)
4 char *filename;
5 IMAGE *image;
6 int level;
7 {
8     readfile(filename, image, 0, 0);
9     threshold(image, image, level);
10    contour(image, image, 0, 8, 0);
11 }

```

Figure 3-16 : Listing code in the worksheet

A full description of these and other commands for program development can be found in Chapter 4.

The second part of this sample session shows how you can make your programs available as commands in a *menu*, and make a menu of your own.

The information used by SCIL_Image to generate menus and dialog boxes resides in a file called 'comfile'. This file is generated from smaller files. These are recognizable by a '.cdf' suffix, and all are stored in the 'scilimage14\prog\comfiles' directory. You can add functions and menus of your own by editing the file 'myfunc.cdf', residing in 'scilimage14\prog\example', and following a procedure we will now describe by an example.

Let us assume that you want your function `myfunc()` to appear as the command **myfunc** in the standard 'Options' menu of SCIL_Image and in a new menu which you want to call 'MyMenu'. Then you should edit 'myfunc.cdf' so that it contains:

```

$MyMenu      $SCILIMAGE

FUNC myfunc
MENU MyMenu Options
OPTIONS NOT_COMPILED
ARGS
    filename - trui - * My File
    image - A - - My Image
    odd - 127 1 255 Only odd values
#

```

Make sure that no empty lines follow the last line of `myfunc.cdf`. In order to add 'myfunc.cdf' to 'comfile',

- 1) Place 'myfunc.c' in one of the directories listed in the environment variable `MACRO` (Chapter 2), such as 'scilimage14\prog\example\myfunc.c'.

- 2) Store the CDF-file 'myfunc.cdf' in a directory of your choice, for instance 'scilimage14\prog\example\myfunc.cdf'.
- 3) Exit SCIL_Image (if it is running) and start the Microsoft Developer Studio (devstudio) by double-clicking the file the workspace file for SCIL_Image:

c:\scilimage14\prog\scilimag.dsw

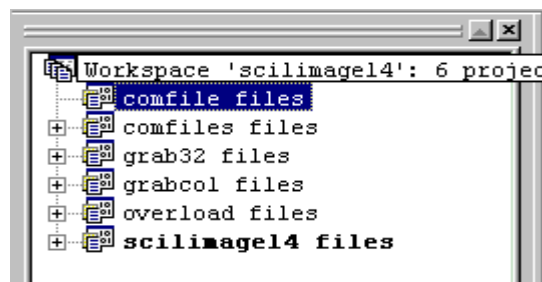
- 4) From within the Developer Studio op the file 'c:\scilimage14\prog\comfiles.mak' **be sure to set the "Open as" option to "Text" otherwise the wrong things happen**). At the end of the COMFILES list add the line:

\$(SCIL_ROOT)\prog\example\myfunc.cdf

(make sure to add it onto the last line in the block COMFILES, and preceding the Return on the previous line by a backslash '\', just as was done on the other lines).

- 5) Save the changed comfile by clicking the save button on the button bar or choosing "File:Save".

- 6) To create the new \scilimage14\comfile, right click the "comfile" project and choose Build.:



- 7) In the output window a list of files must now be printed (if not, the cdf-file you added was older than the comfile and must be saved again).
- 8) Start SCIL_Image again (either by hitting <CTRL>F5 in the devstudio or by double-clicking its program icon as you did before).

As you see, the 'MyMenu' button is now present in the SCIL_Image menu bar. The command **myfunc** is now an item of 'MyMenu' as well as an item of the 'Options' menu. Let's try it:

select 'MyMenu:myfunc'

A dialog box pops up for your personal command, which you may execute by clicking 'OK'.

The function `myfunc()` is not yet a *compiled* function; it is still loaded from the file 'myfunc.c' and then *interpreted*. It is possible to add the function to the library of compiled functions of SCIL_Image and so make it into a full-fledged command, but explanation of that will have to wait till Chapter 5.

This concludes the fourth sample session.

Making a New Compiled Version of SCIL_Image

In this sample session we will show you how to embed your functions as commands in a new *compiled* version of SCIL_Image. We take a simple example in image processing. At first, this example shows you how to obtain pointers to images - then we extend it to use the convenient error checking functions of SCIL_Image. Finally, we extend it once more to make *one* command that treats images of various types (in this case integer-valued and real-valued images).

We will write a program for a command that adds 1 to an image - not very useful, but it illustrates the issues involved. We enter it using the 'File:New C Program' menu selection:

```
/* add_one.c : first version */
#include "image.h"

g_add_one (IMAGE *in, IMAGE *out)
{
    PIXEL    *indata, *outdata;
    long     num;

    indata   = ImageInData(in);
    outdata  = ImageOutData(out);
    num      = ImageSize(in);

    while (--num)    *outdata++ = *(indata++) + 1;

    display_image(out);
}
```

Note that we do not need to start with `main()`, since SCIL_Image will be the main program that will call this function. Thus the terms 'program' and 'function' are almost synonymous. We use 'program' for the contents of the file, including the `#include` statement and possible other functions. A 'function' is then any of the function bodies in the program. The 'main function' is the function `<any_name>()` defined in the file '`<any_name>.c`'.

The program itself looks much like you would expect from a C-program. The include file 'image.h' contains the definitions of the `IMAGE` and `PIXEL` data types, and also of the pre-processor macros `ImageInData()`, `ImageOutData()` and `ImageSize()`. The use of these macros is explained in the chapter "Programming with Image" - for now, know that the first two return the pointers to the input and output images, and the third returns the size of an image.

In itself, this function can be used, but it is not yet properly embedded within the SCIL_Image structure. If you execute it, you will need to call the function `display_image()` explicitly, to display the image. To embed the function properly into SCIL_Image, we add some lines:

```
/* add_one.c : version 2 : pre_op, post_op added */
#include "image.h"
#include "im_infra.h"

g_add_one (IMAGE *in, IMAGE *out)
{
    PIXEL    *indata, *outdata;
    long     num;

    if ( !pre_op(in,out,ADJUST,G_2D_SPEC,GREY_2D) )
        return( NOT_OK );

    indata = ImageInData(in);
    outdata = ImageOutData(out);
    num     = ImageSize(in);

    while (--num)    *outdata++ = *(indata++) + 1;

    return post_op(out);
}
```

The functions `pre_op()` and `post_op()` do the administration involved in making sure that your function `g_add_one()` works on images of type 'g2D'. They will pop up error messages if you try to use the function in ways you did not specify. If your operation could not be performed 'in place' (from an image to that same image), they would create (`pre_op`) and destroy (`post_op`) appropriate intermediate images. The function `post_op()` also takes care of the automatic display of the result. These functions are described in the chapter "Programming with Image".

Now you have made a function that can be put into SCIL_Image - but it will only add 1 to images of type 'g2D', which are integer-valued. If somewhere in other calculations you would have made a real-valued image, you would not be able to add 1 to it using this function. On the other hand, you would not want to have an 'add 1'-function for every possible image type (you would have to be aware of what image type you would want to process before you could choose the appropriate function!).

SCIL_Image has a way to prevent this - you can *overload* a command. This means that the command itself decides what functions to call, dependent on the image types you give as arguments. We apply this to extend our example to treat real-valued images as well, in one overloaded command **add_one**. For that, we need to define a function `f_add_one()`, which adds 1 to a float image. It is very similar to `g_add_one()` (see below in the box).

We also define one main function, `add_one()` which will select either `g_add_one()` or `f_add_one()`, depending on the type of the input image `in`. Its name must correspond to the name of the command we are constructing (**add_one**). The overloading involves a function `overload_func()` which can perform that selection on the basis of data we have given it. Change the file once more to correspond to the box below (the function `g_add_one()` is unmodified, and not shown fully), and save it.

```

/* add_one.c 3rd : overloading the operation */
#include "image.h"
#include "im_infra.h"

int (*func)(IMAGE *, IMAGE *);

add_one (IMAGE *in, IMAGE *out)
{
    func = overload_func("add_one", in);
    if ( !func) return (NOT_OK);
    return (*func)(in,out);
}

f_add_one (IMAGE *in, IMAGE *out)
{
    float *indata, *outdata;
    long num;

    if ( !pre_op(in,out,ADJUST,F_2D_SPEC,FLOAT_2D) )
        return(NOT_OK);

    indata = ImageInData(in);
    outdata = ImageOutData(out);
    num = ImageSize(in);

    while (--num) *outdata++ = *(indata++) + 1.0;

    return(post_op(out));
}

g_add_one (IMAGE *in, IMAGE *out)
{

```

SCIL_Image still needs to be told that the command name **add_one** has been overloaded to floats and integers. To do so, make a new file 'add_one.ovl', in the directory 'scilimage14\prog\overload'. The contents should be:

```

# add_one.ovl: overload file for add_one
#
TABLE    g2d    GREY_2D    G_2D_SPEC    2
        add_one    g_add_one
#
TABLE    f2d    FLOAT_2D    F_2D_SPEC    2
        add_one    f_add_one
#

```


Such a file is called an *overload file* (its syntax is explained in the chapter "Programming with Image"). You only need to make an overload file for commands that should be able to handle various image types.

To make the command **add_one** usable under SCIL_Image, and bring it under the menu, we make a file in 'scilimage14\prog\comfiles'. This file is similar to the CDF file we saw in Sample Session Three:

```
# add_one.cdf: function description
#
FUNC      add_one
MENU      Arithmetic
ARGS
  image - A - - Input Image
  image - B - - Output Image
#
```

It determines the default values of the images, and helps to pop up the appropriate dialog boxes.

We now have all ingredients to make a new version of SCIL_Image that contains our **add_one** command. We still have to compile the file 'add_one.c', and to make a new version of SCIL_Image.

- 1) Open the Microsoft Developer Studio workspace "c:\scilimage14\prog\scilimag.dsw" (if not already open) and add the file add_one to the project "mylib" (right click "mylib" and choose "Add files to project" and then select the file add_one.c from the directory you put it in.)
- 2) Open de file "c:\scilimage14\prog\comfiles.mak" (set the "Open as" to text in the dialog box) and at the end of the COMFILES list add:

c:\scilimage14\prog\example\add_one.cdf

and save the file. (fill in the appropriate path name if you saved the file "add_one.cdf" somewhere else).

- 3) Open de file "c:\scilimage14\prog\overload.mak" (set the "Open as" to text in the dialog box) and at the end of the OVERLOAD list add:

c:\scilimage14\prog\example\add_one.ovl

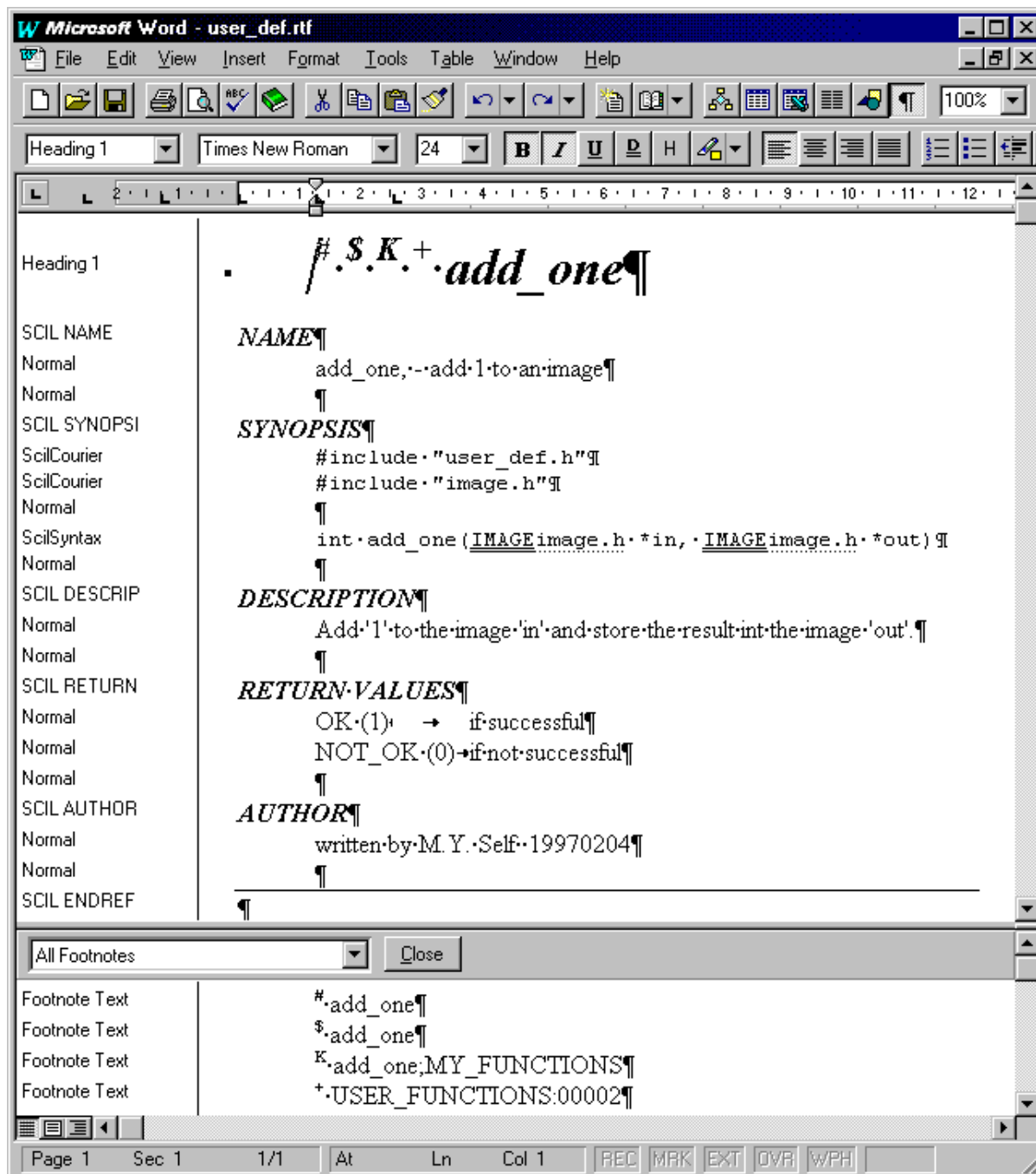
and save the file. (fill in the appropriate path name if you saved the file "add_one.ovl" somewhere else).

- 4) For both the "comfiles" and "overload" project, choose the "Build" command on the context menu (when clicking the project with the right mouse-button).
- 5) Then rebuild SCIL_Image by choosing the "Build" command on the "scilimage14" project.
- 6) Start SCIL_Image again. In the "Arithmetic" menu you will find your command "add_one".

Please note that just must build the "comfiles" and "overload" projects by hand first because Developer Studio does not recompile the files "sysfunc.c" and "overload.c" when necessary.

To test it, do not read in an image, just choose your new command, either by typing or by selecting it from the 'Arithmetic' menu. After your command has been executed (this is almost immediately) you may check that it indeed added 1. Clicking with the cursor in image window A shows that it has value 0, and clicking in window B should show that it has value 1. Now convert image A into a real-valued image (by selecting 'Image:Conversion:convert') and repeat your command. Now it changes values of 0.0000 to 1.0000. Thus it works on real-valued images as well.

Now that you know that the command works, it is time to add a description of it to the on-line Help facility. You should do this by following the instructions on the Help page: '**User-defined Help Pages**'. It requires the editing, in Rich Text Format, of the file '**user_def.rtf**', located in the 'help' directory. This file contains a sample manual page that you can use to create your own manual pages. For this you need an editor that can read this format. After you are done composing your Help page, according to the instructions, your editor window should look similar to this:



Save it, and start the "Help Workshop" from the "Microsoft Visual C++" environment. In the Help Workshop open the file 'scilimag.hpj' (from the help directory). To compile the help file:

Select 'File:Compile'

Compilation will take some time, the help compiler will minimize itself during compilation and restore itself to normal view after compilation. It generates the new 'scilimag.hlp', which makes your Help page reachable from SCIL_Image like that of any other command:

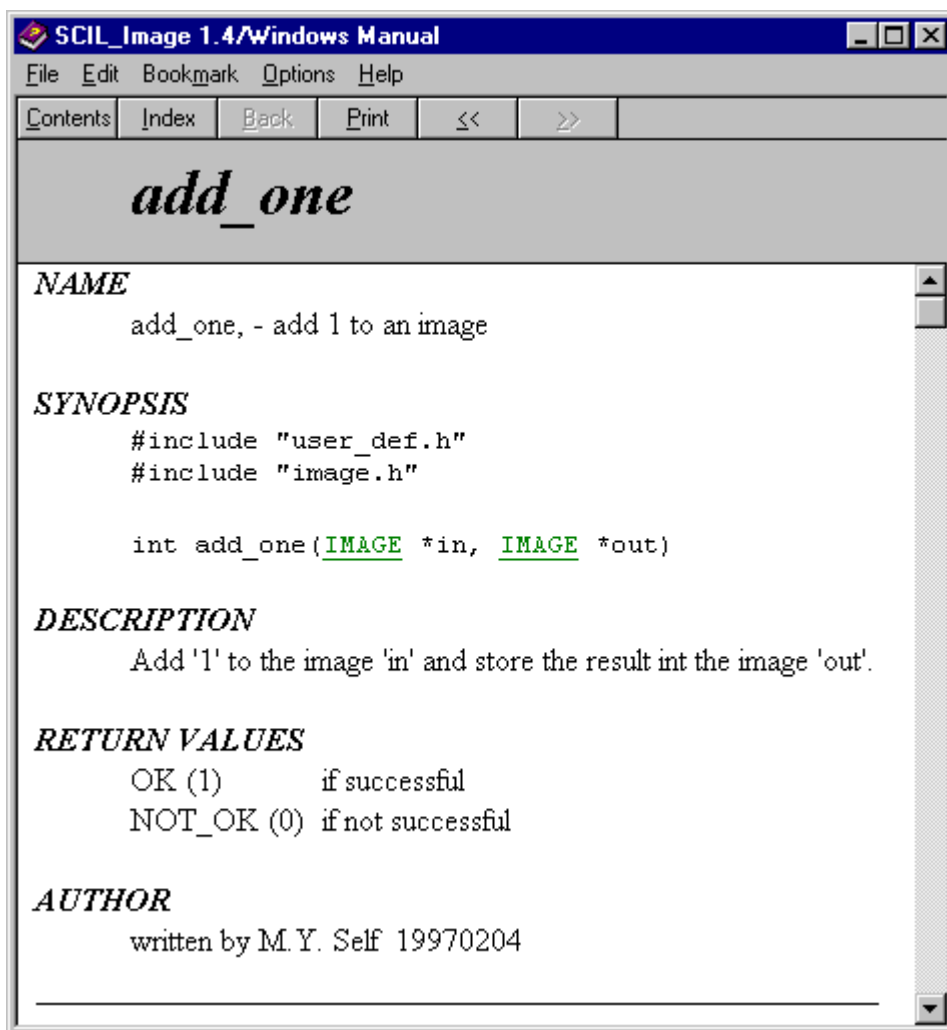


Figure 3-17: Your new manual page

We have used the example to show you that it is possible to embed your own function as a command in SCIL_Image, rather than as an exact explanation of all steps involved. We will give that in Chapter 5. You should remember these salient points:

- You can embed your own functions in SCIL_Image in a way that makes them indistinguishable from the standard functions - they are then indistinguishable from the standard SCIL_Image commands. This means that you can make your own version of SCIL_Image, completely tailored to your own application.
- You should use `pre_op()` and `post_op()` in your image processing programs to have the advantage of SCIL_Image's type checking, error handling and automatic displaying of results.
- You can overload one command name to treat images of various types, using the overload mechanism.

This concludes the fifth session..

The Commands of the SCIL_Image Menus

In this section, we briefly describe the menus in the SCIL_Image menu bar. This will give you an overview of SCIL_Image's capabilities. We will specify the sections in which you can find more detailed descriptions of the commands.

File

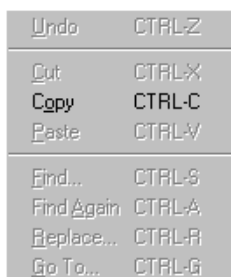


New C Program	CTRL-N
Open C Program...	CTRL-O
Open Image...	
Save	F2
Save As...	
Save Image As...	
Run	F12
Print...	CTRL-P
Printer Setup...	
Print Image Setup...	
Exit SCIL	
Q C:\scilimag\PROG\EXAMPLE\myfunc.c	

Figure 3-18 : the File menu

This menu contains commands for file manipulation. Most commands are standard. SCIL_Image also provides the 'Open Image' command, to open any of the image files.

Edit



Undo	CTRL-Z
Cut	CTRL-X
Copy	CTRL-C
Paste	CTRL-V
Find...	CTRL-F
Find Again	CTRL-A
Replace...	CTRL-R
Go To...	CTRL-G

Figure 3-19 : the Edit menu

This is the standard editor menu. You can use it to edit text in any text window, or to move text between windows.

SCIL

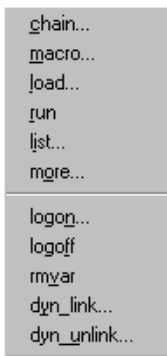


Figure 3-20 : the SCIL menu

The 'SCIL' menu contains commands necessary to the functioning of SCIL_Image as a programming environment. These commands are described in detail in "The C Interpreter" (chapter 4).

chain, macro, load, run are commands to manipulate interpreted programs in SCIL_Image.

list, logon, logoff, rmvar are commands used for program development, to show and record interpreted programs in SCIL_Image.

Image

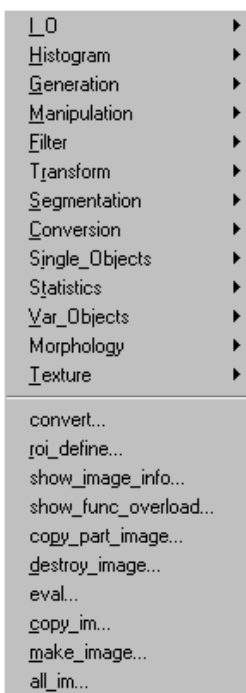


Figure 3-21 : the Image menu

This menu contains the image processing commands, and is therefore the menu you will probably use most. It contains many sub-menus.

I_O contains functions which read and write images from and to your disk.

Histogram has commands to manipulate image histograms.

Generation has commands to generate test images, with simple geometric and grey-value patterns, to test your algorithms.

Manipulation contains operations that 'move pixels around', such as reflections and rotations of images.

Filter contains a standard library of filters, both linear (for instance, the uniform and the laplacian filter) and non-linear (for instance, Roberts' gradient and the median filter).

Transform has some transformations.

Segmentation contains various thresholding algorithms.

Conversion allows you to convert between different types of images.

Single_Objects allow you to reduce your processing to single objects in an image, for instance to determine their convex hull, or to perform specific measurements.

Statistics has commands that compute some simple image statistics, such as mean value, minimum and maximum.

Var_objects contains the manipulation of 'var_objects', which are non-image data structures akin to arrays in C.

Morphology contains many operations from *mathematical morphology* such as erosion, dilation, skeleton, and hit-or-miss transforms. Some frequently used operations have received special attention in the implementation, including the 3x3 erosion and dilation. If you are a specialist in mathematical morphology, note the options for special points in skeletons, and mathematical morphology with arbitrary shaped elements.

Texture has commands that deal with texture (measurements).

The commands in the remainder of the 'Image' menu, can be split into two groups often used in combination with image processing commands:

Image creation commands are a group of commands that help you create and destroy new images, and regions of interest in images.

Various image utilities contains some commands that occur in other menus as well, but are often used in combination with image processing commands. It also has the important `copy_im` command to copy images.

Display

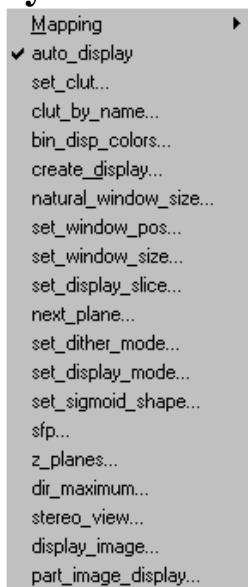


Figure 3-22 : the Display menu

This menu contains commands that manipulate the display of your images. The commands can affect size, location, grey-values and colors of the images.

Options

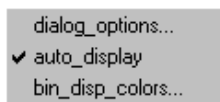


Figure 3-23 : the Options menu

This menu permits you to tailor the interaction of SCIL_Image (dialog boxes, display, reaction on mouse-clicks in images, etc.) to your taste.

Arithmetic



Figure 3-24 : the Arithmetic menu

This menu contains various commands to perform arithmetic operations on images. They are divided into three groups:

Variable arithmetic operations (Int_based - Exotic) perform computations on SCIL_Image variables.

Image arithmetic operations (add_im - clip) perform arithmetic computations on image pixels, point-wise for the complete image (this allows you to divide one image by another image, for instance)

Arbitrary evaluation consists of the very useful command **eval**, which evaluates C-like expressions with images and image coordinates.

Itools

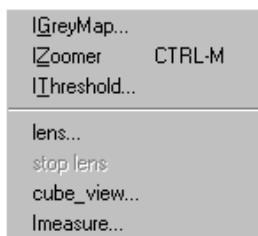


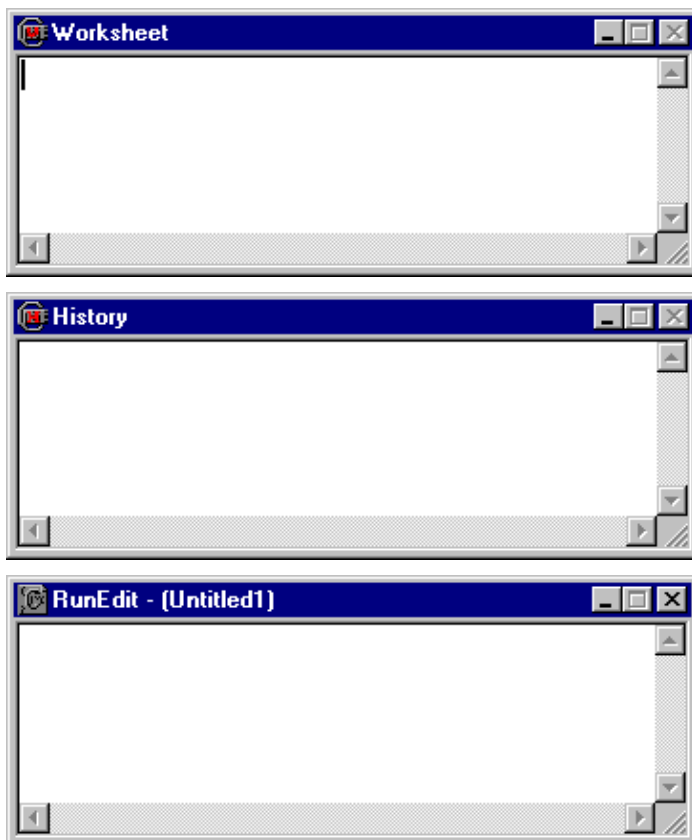
Figure 3-25 : the Itools menu

This menu contains some interactive tools. We described them in "**Session One: Viewing Images**"

The Properties of Text Windows

In the sample sessions, you have encountered various text windows and their properties. Here is a summary.

There are 3 types of text windows in SCIL_Image. They are: the Worksheet, the History window, and any number of RunEdit windows:



The following key sequences can be given in any text window, after selecting a command or a number of commands. Their effect in all windows is the same.

Enter (on main keyboard) gives a new-line.

Enter (on numeric keypad) **or Shift-Enter** (on main keyboard) executes a selection, and reports that execution in the Worksheet and the History window. If no command has been selected, the line on which the cursor presently resides is executed.

Ctrl-Enter pops up a dialog box for the selection. By clicking **Help** in that box, you obtain on-line Help information.

You will find these special key sequences very convenient when developing your applications.

On-line manuals

The documentation of SCIL_Image 1.4 is supplied in Adobe PDF format. To read these documents the Acrobat Reader (supplied with SCIL_Image) is needed. The "Help on SCIL_Image" button from the Help menu produces a window as shown below.

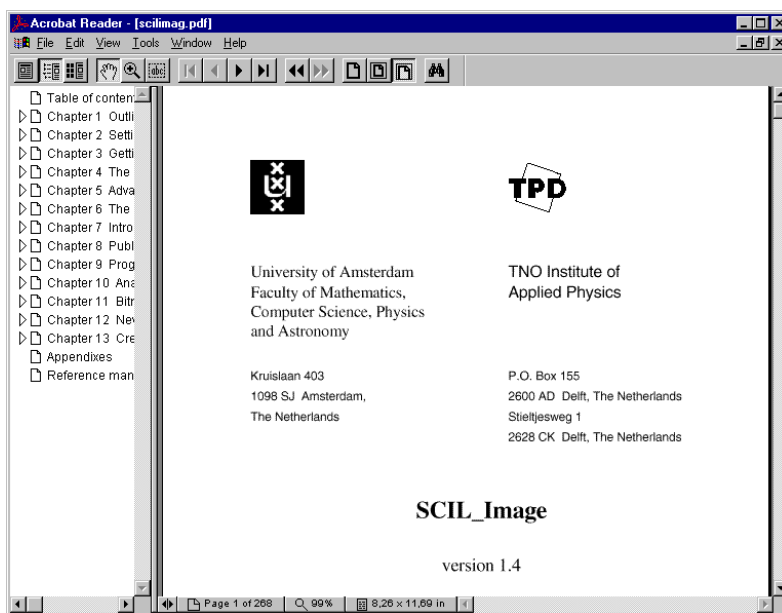


Figure 3-26: the acrobat reader

Navigating through the manual can be done in several ways, using the cursor-keys, the scrollbars, the buttons in the toolbar or clicking the bookmarks in the left pane. Extended help on using the reader is given in the Help menu of the reader itself.

Reference manual

The reference manual opens as shown below, at the index page.

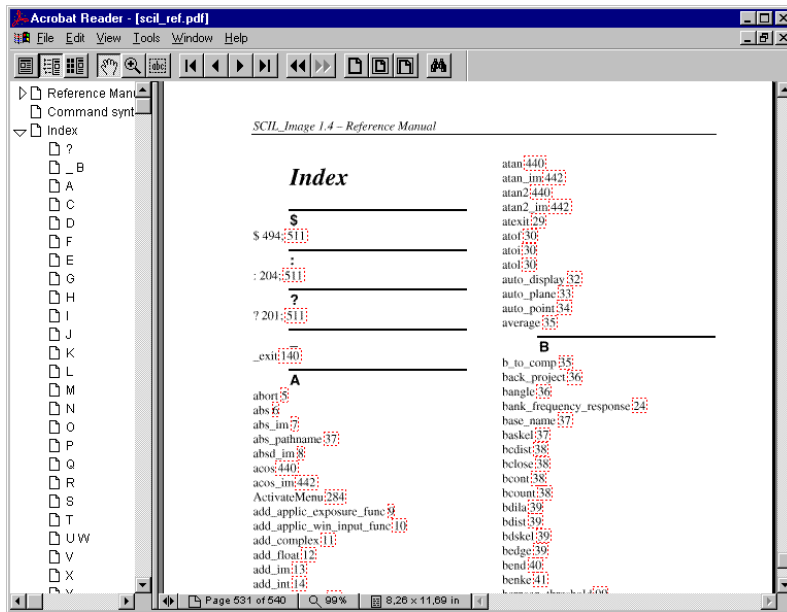


Figure 3-27: the reference manual index

To find the reference page of a function, search for the function in the Index and then click on the page number behind it (within the red square). The reader then jumps to the page on which that function is described. As shown below with the `readfile()` function.

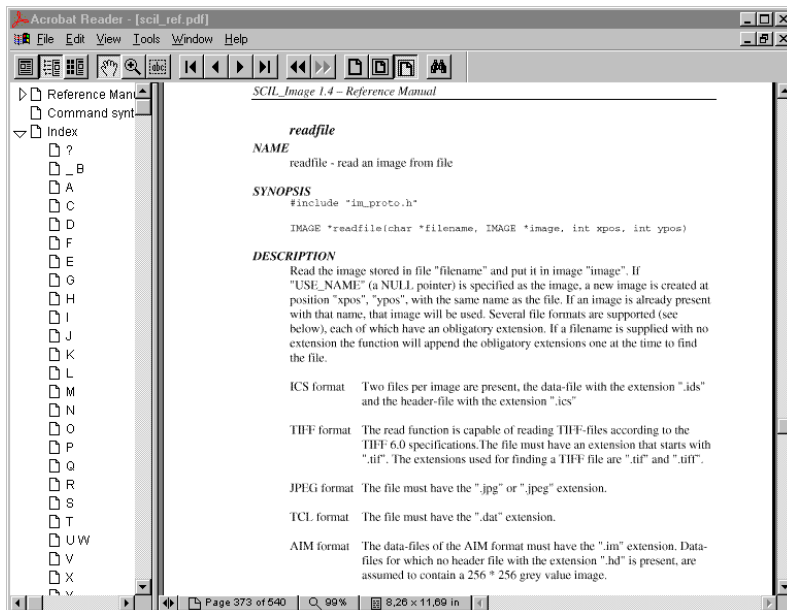


Figure 3-28: a reference manual page

Chapter 4 The C Interpreter

When you write programs in a high-level programming language like C, you have to follow the sequence edit-compile-load-run. With SCIL_Image, it is possible to just edit and run your program. The part of SCIL_Image that makes this possible is called the C interpreter. In this chapter we discuss the use of this interpreter for developing software. You *can* compile and load if you want to make your programs run faster (consult "Advanced SCIL_Image" on page 5-1).

Read this chapter if:

- You have experience with the C programming language.
- You are experienced with the menu and command modes.
- You want information on the use of macros.
- You want to combine image processing commands with C control statements.
- You want to develop image processing programs.
- You need to know the limitations of SCIL_Image's C language.

Do not read this chapter if:

- you have no experience with SCIL_Image.
- you have never programmed in C.
- you only want to use SCIL_Image at the Menu Mode of interaction.

SCIL_Image and C

After starting SCIL_Image, you can enter any SCIL_Image command or C statement into the Worksheet. It will be *interpreted* immediately after you hit the 'Enter' key. *Interpreted* means that SCIL_Image:

- Identifies the C-code that will execute your command
- Fills in missing arguments with default values
- Executes the C-code and displays the results

The first two steps are called *command expansion*. This feature of SCIL_Image means that you do not need to use the exact name or arguments of the C-codes that do image processing. For instance, you can just type **readf trui**, without knowing that this is actually the C-statement **readfile ('c:\scilimage14\images\trui', A, 300, 300);**.

SCIL_Image is also capable of interpreting *C-statements*. This means that if you want to execute a command 10 times, you can type **for(i=0; i<10; i++) <command>;** because SCIL_Image understands the control flow of standard C. A sequence of commands to SCIL_Image may therefore look very much like C-code. You can collect such sequences in files, and execute them. Such files are called *macros*.

SCIL_Image can also understand actual C-programs and run them without compiling. We call these programs *interpreted programs*.

In this chapter we explain these three ways in which SCIL_Image interprets C, as well as some basic commands that help you develop your programs.

However, a warning before you start:

SCIL_Image's C language is close to Kernighan & Ritchie 1978 C¹, but there are some limitations (see "Features of SCIL_Image's C-interpretor" on page 4-16).

¹ Kernighan B. W., Ritchie D. M., The C Programming Language, PRENTICE-HALL, INC., Englewood Cliffs, New Jersey, ISBN 0-13-110163-3.

ANSI-C compatibility

Although the C language of SCIL_Image is not ANSI-C, we strongly recommend that ANSI-C coding is used whenever **compiled** (not interpreted) code is added to SCIL_Image. The function-calling mechanism of the C-interpreter has been changed to the ANSI-C calling convention. Consequently, adding compiled K&R C-code, **may** cause the arguments to be passed incorrectly to those functions.

The Direct Command Mode

You are in the *Direct Command Mode* when you type commands into the Worksheet. As you have seen in Sample Session Three (page 3-21), you can use the Worksheet to give SCIL_Image commands like:

readf trui

(followed by hitting the 'Enter' key on the numeric pad). Commands like these are expanded internally to call the C-function `readfile()`. This command has four arguments. SCIL_Image fills in the default values to make your command **readf trui** into a proper function call. You can see what these default values are by typing

readf ?

or by popping up a dialog box:

type 'readf', select it by double-clicking, and hit 'Ctrl-Enter'

Thus the proper C-statement corresponding to 'readf trui' is:

readfile("trui",A,300,300);

All commands in SCIL_Image have these three versions:

<i>SCIL_Image menu selection</i>	select 'Image:I/O:readfile'
<i>SCIL_Image command</i>	readf trui
<i>C function call</i>	readfile ("trui",A,300,300);

You should bear in mind the differences between these terms. We call the last two possibilities the *Direct Command Mode*, because you type them into the Worksheet.

Commands in the Direct Command Mode are very much like *lines* in a C program, both in syntax and in meaning. You can combine several commands on one line using the same rules as in C. More specifically, the syntactic possibilities for commands given in the Direct Command Mode are:

- **Classes of commands**

- **Variable declarations:**

```
int i;
```

- **Statements:**

```
for(i=0;i<3;i++) printf("%d\n", i);
```

- **File inclusion directives:**

```
#include "image.h"
```

- **Pre-processor defines:**

```
#define max(a,b) (((a)>(b))?(a):(b))
```

- **Type definitions:**

```
typedef struct date_t {int month,year;}DATE;
```

```
DATE date;
```

```
date.month = 2; date.year = 1993;
```

- **C-functions**

In SCIL_Image, functions are invoked in the same manner as in C. The functions used throughout this section are part of the standard C library. You cannot define your own functions in the Direct Command Mode (you would always have to define those using an editor because a program is a named collection of commands). There are ways to do this in the other modes. Once you have defined your own functions, you can use them in the Direct Command Mode if they are:

- *macros* (see "**The Macro Mode**" on page 4-5)
- *loaded interpreted functions* or *UFOs* (see "**The Programming Mode: Interpreted C-functions and UFOs**" on page 4-7)
- *compiled C functions* (Chapter 5)

- **Putting commands together**

More than one direct command may be entered on the same input line, if separated by semicolons (;). For example:

```
readf trui; printf("There she is again !\n");
```

The first semicolon is thus a separator between direct commands, the second semicolon has to be given since the second command is a C-function call, and is not syntactically correct without it.

Using the Worksheet and the History window, you can work very flexible in the Direct Command Mode, as our 'Sample Session Three' (page3-21) showed. Both the Worksheet and the History window show previously entered commands. The difference is that the History window cannot be edited, it retains an exact account of what you have done. But the Worksheet can be edited, which is convenient if you want to type a similar command to one you did before. It uses the standard Windows editor, available under the Edit menu. Both windows can be resized, moved and scrolled. We repeat the main possibilities of these windows:

- If you want to *execute more than one command* in the Worksheet or in the History window, you select them by using the click-and-drag method until all desired commands are highlighted; then execute them - hit the 'Enter' key on the numeric keypad, or 'Shift-Enter' on the keyboard..
- If you want to *re-execute a command*, you can do this simply by moving the cursor to that line in the Worksheet or History window, clicking and executing.
- You can also *edit a command* in the Worksheet (not in the History window) using the normal method (mouse selection and re-typing), before you re-execute it. In the same way, you can remove lines.

The above even applies to the RunEdit window: mouse selection and 'Enter' key executes your selection as if it were in the Direct Command Mode.

The Macro Mode

After you have worked in the Direct Command Mode for a while, you will notice that some command sequences have a tendency to re-occur in your work. You can of course type them in once and re-execute them using the Worksheet and/or History windows, but SCIL_Image provides an easier way. You can make a file that contains such a sequence of commands, and execute the file as a whole. Such a file of direct commands is called a *macro*.

To make a macro, do the following:

- 1) Go into the RunEditor, using 'File:New C Program' under the menu system
- 2) Type in the desired sequence, or copy it from the Worksheet or the History window using the mouse-select and the commands in the 'Edit' menu.

- 3) Save the file (using 'File:Save') in one of the directories listed in the environment variable `MACRO` (see Chapter 2), for example in 'c:\scilimage14\demo\macro'. As a filename, use a name of your choice followed by the suffix '.mac' (for macro).
- 4) Exit the RunEditor by clicking the close box of its window.

To run your macro file, use the **macro** command, which has the following syntax:

```
macro [-i] [-v] <macrofile>
```

The **macro** command executes the lines in the macro file as if they were typed in one by one. The contents of macro files are restricted to lines with the syntax of the Direct Command Mode. A macro file is *not* a function: it can have no arguments.

As an example, suppose it is frequently necessary to clear four standard images named A, B, C, D. Instead of typing the command **clear_im** four times, a macro file can be created. Edit a file (using 'File:New C Program') to contain:

```
clear_im A
clear_im B
clear_im C
clear_im D
```

Save the file (using 'File:Save') as 'cleanup.mac' (recall that the suffix '.mac' in the filename is meant to indicate a macro file). Now we have created a macro that clears all images. First we read an image file into each image window.

```
readf trui a; copy_im a b; copy_im b c; copy_im c d
```

Then we clear them, using our newly created macro:

```
macro cleanup.mac
```

The **macro** command has two optional arguments:

- i Each line is executed with user interaction
- v Each line is printed to the Worksheet window ('verbose')

When you use the **-v** option, please note that the individual commands from the macro appear in the Worksheet, but that the History window only contains the **macro** command.

The Programming Mode: Interpreted C-functions and UFOs

To use SCIL_Image in the *Programming Mode* (rather than the Direct Command Mode) means that you use an *editor* to create or change a program, after which you load the program into SCIL_Image and run it. This process may be repeated several times, just as in ordinary programming. The basic development scheme is : edit-load-run-edit,- etc. The program may consist of #includes, #defines, global variables and functions with local variables, just as in standard C. This allows you to define C-programs for the SCIL_Image C-interpreter. There is no need to compile them.

To make an *interpreted* (so non-compiled) C-program, do the following:

- 1) Create a file using the editor. You may use SCIL_Image commands, but we advise against this, since you would have to redo those commands if you were to decide later that you wanted to make the functions in the program into compiled functions (see "**Advanced SCIL_Image**" on page 5-1). Please note that almost all standard C syntax is permitted, the only exceptions can be found in "Features of SCIL_Image's C-interpreter" on page 4-16.
- 2) Save the file in a directory listed in the environment variable MACRO (Chapter 2). You should choose a name with the format '`<name>.c`', to distinguish the program from a macro file.
- 3) Run the program. There are three ways to run it:
 - From the menu by:
select 'File:Run'
(only if your program has a `main()!`)
 - Type in the Worksheet:
load <name>.c
run
 - Type in the Worksheet:
chain <name.c>

These three commands **load**, **run** and **chain** are also available under the 'SCIL' menu.

We demonstrate this by an example. Type it and save it in a file which you name '`simpprog.c`'.

```
/* simpprog.c */
#define STRINGSIZE 80

char string[STRINGSIZE];

main()
{
    int i = 0;
    for (;i<2;i++) {
        printf("Enter a string :");
        gets(string);
        printf("The number of characters is %d\n",
            charcount(string));
    }
}

int charcount(s)
char *s;
{
    int cnt = 0;
    while (*s++ != '\0') cnt++; return(cnt);
}
```

Now run it:

select 'File:Run'

Enter a string : WE NOW TYPE A STRING

The number of characters is 20

Enter a string : THAT WAS IT !

The number of characters is 13

After a program is loaded, any of its functions may be called from the command line. For example:

load simpprog.c

printf("%d\n", charcount("How many characters am I"));

24

However, during a SCIL_Image session, you may want to load and run a new program. *This will delete all definitions (made in your old program) of variables and functions from SCIL_Image's working memory.* The two commands that load a new program and cause this loss of memory are **load** and **chain**. Sometimes you want to clear the working memory - the command **rmvar** does this. But remember:

BEWARE OF rmvar, load, chain !

since they always clear SCIL_Image's memory.

This does *not* mean that you have to include all your functions in every new program that you write (since otherwise they might not be known), because SCIL_Image offers the *UFO*. This means *Undefined Function Obtainer*, and describes SCIL_Image's capability to find non-compiled user functions. The functions not defined in a program are automatically loaded into SCIL_Image's memory, provided that:

- Each undefined function is defined in a file with the same name followed by the suffix '.c'.
- Each file can be found in the current directory or in one of the directories in the environment variable MACRO.

Such a function is retrieved automatically and executed. Like your program, the UFO will stay in SCIL_Image's memory until you give **load**, **rmvar**, or **chain** command. When a **list** command is given, both the text of the loaded program and the text of the UFOs are shown.

Program Development Commands

In this section, we give the commands which can be entered from the Direct Command Mode, and which are used for program development. Some of these commands have been discussed before, others are new. Here is a summary:

chain <file> [args]	load and run program
list [start],[end]	show program text
load [file]	load program text
logon <file>	connect logfile to session
logoff	disconnect logfile
macro [-i] [-v] <file>	execute macro file
rmvar	remove old variables and free allocated space
run [args]	interpret loaded program (including main(!))
time <command>	time a command
<pause/break key>	interrupt program execution
? <pattern>	list functions with names matching <pattern>
<command> ?	prompt for parameters of command
<Ctrl-Enter>	pop up dialog box of selected command
<Ctrl-H>	pop up Help facility

chain <filename> [args]

The **chain** command loads the file and starts running it immediately. So instead of:

```
load example.c  
run
```

you can just type:

```
chain example.c
```

Please note the difference between the **chain** command and the **macro** command:

```
chain :      loads file with a fully correct C-syntax (a C-program)  
macro :      loads file that contains direct SCIL_Image commands (a macro)
```

The **chain** command is found in the SCIL menu as 'SCIL:chain'.

list [start],[end]

The text of the loaded program can be displayed with **list**.

```
rmvar  
#include "image.h"  
list 5           Displays line 5 of program text  
list 5,10       Lists lines from 5 to 10  
list ,10        Lists from start to line 10  
list 10,        Lists from line 10 to the end
```

load <filename>

The **load** command loads program text into SCIL_Image. The size of a program is only limited by the amount of RAM available. When no filename is entered, SCIL_Image will prompt for one. The following command loads the file 'demo.c':

```
load "demo\meas.c"
```

Filenames are arbitrary, but it is advisable to use some sort of convention: here we always use the suffix '.c' to indicate files containing commands. When loading a specific file the contents of the previously loaded file are overwritten and the global variables are removed. Directly after loading a file, the functions in the file are known to the C-interpreter. They can then be

used in the Direct Command Mode, provided that they do not use any of the global variables in the file, since these will only be declared when the entire program is run.

logon <filename> **logoff**

The **logon** command followed by a filename creates a file in which all direct commands given during a session are stored. Later, this file can be used as a macro. In the following example, the file `last_session` is opened and subsequent commands are stored in it. The **logoff** command is used to discontinue command logging and close the file.

```
logon last_session  
printf("Good night, Irene\n");  
Good night, Irene  
logoff  
macro last_session  
Good night, Irene
```

If a logfile is active, and another **logon** command is issued, the active logfile is closed, and subsequent commands will be stored in the file specified in the most recent **logon**.

Why do you need **logon/logoff** if you already have a History window ? If you always give innocuous commands, there is no need: all commands you give are recorded faithfully in the History window, and whenever you wish you can use an editor to extract the commands you want to keep and store them in a file. However, should you give a command that gets SCIL_Image 'stuck' or makes it 'crash', then you will have no such opportunity. In that case, **logon** would have saved all you did in a file, so you could salvage what is valuable, or trace your mistake. Therefore, if you are developing a complicated application, using **logon** might be a safe precaution.

macro [-i] [-v] <macrofile>

The **macro** command executes the lines in a macro file as if they were typed in one by one. Therefore the lines of macro files are restricted to commands you would give in the Direct Command Mode.

- i With the **-i** option every line is executed interactively: each individual command from the macro file will appear on the screen, followed by the prompt *[y/n/q]*. Entering **y** or pressing **<return>** will execute the command, entering **n** will skip the command and **q** will exit the file and returns you to the SCIL_Image Direct Command Mode.
- v With the **-v** option, every line is written to the Worksheet window as it is executed. The lines are not written to the History window - just the **macro** command itself is.

Please note the difference between the **macro** command and the **chain** command:

macro : loads file that contains direct SCIL_Image commands (a macro)
chain : loads file with a fully correct C-syntax (a C-program)

rmvar

The **rmvar** command clears the C-interpreter's memory - it removes the old programs, including all its functions, variables, structure descriptions, type definitions and pre-processor defines. Note that variables, defines, and typedefs are also removed whenever a **load** or **chain** command is given.

rmvar
list

run [args]

The **run** command (optionally with arguments) is used to execute a previously loaded file. The **run** command first declares all global variables and then calls the function `main()` with the specified arguments (if any).

time <command>

The **time** command records the time it takes to execute the specified command. This is quite useful for benchmarking your commands. For example:

```
int t = 1000;
time while( --t );
time: 0.250
```


the interrupt: Pause/Break

To interrupt a running program:

hit 'Pause/Break'

This indicates to SCIL_Image that the execution of the current program should be stopped. SCIL_Image will then return to the Direct Command Mode. The status bar indicator changes from 'Running' to 'Ready'.

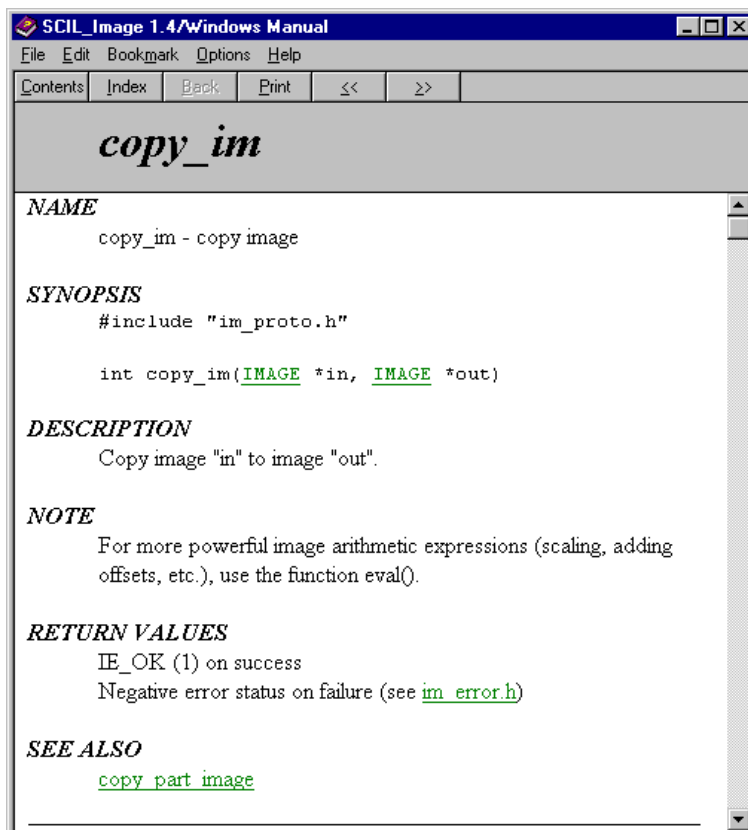
help facilities:

Ctrl-H <selection>Ctrl-Enter <command> ? ? <pattern>

In SCIL_Image, information on commands or functions can be obtained in various ways, and to various extent:

Ctrl-H

pops up the on-line Help facility for SCIL_Image. You can find help from the 'Contents' page, or by using the 'Search' menu. An example of a **H**p page is that of 'copy_im':



The information is in a format similar to that of a C-function on a UNIX system, which is rather self-explanatory even if you have not seen it before. The green underlined terms provide links to other, related help pages. Sometimes you can obtain a popup explanation of terms that are dashed-underlined.

You can also add help pages about *your own functions* and add them to the standard Help facility. You do this by the procedure given on the Help page 'User-defined commands'. They are then reachable in the same way as the functions we put in.

Ctrl-Enter on selection

From any text window, you can pop up a dialog box of a command by

**selecting the command using the mouse (double-click on it)
pressing 'Ctrl-Enter'**

Once on the dialog box, selecting 'Help' will pop up the Help page for the command. This is the more direct way to get the above Help page.

<command> ?

prompts you for the parameters of a command, while offering the default values (do not forget to put a space between the command name and the question mark!):

copy_im ?

Input Image <A/B/C/D> [A] :

Output Image <A/B/C/D> [B] :

? <pattern>

list all commands with names matching <pattern>. The pattern may contain a wild card '*', which can match any sequence of letters and/or symbols.

?s*p

FUNCTION(S) :

set_aio_disp

sfp

silto_comp

sleep

start_comp

strcasecmp

strcmp

strncasecmp

strncmp

The '`?<pattern>`' help option also works on interpreted functions which you may have loaded in macros, and gives some basic information For the interpreted program '`simpprog.c`' defined in **The Programming Mode: Interpreted C-functions and UFOs**" on page 4-7, we have:

```
load simpprog.c
```

```
?charcount
```

```
FUNCTION(S) :
```

```
charcount    Interpreted function. Return type : integer.
```

Errors, Warnings and Diagnostics

The SCIL_Image C interpreter will complain if the syntax of a command is incorrect. The diagnostics are intended to be self explanatory. For instance, forgetting a semicolon will yield:

```
int i
```

```
int i--> syntax error
```

However, SCIL_Image always tries to interpret your statement if it is unambiguous, even if it is not syntactically correct. In this case, it actually declares the variable `i`, despite the incorrect syntax. You can see this by trying it again, now with the semicolon:

```
int i;
```

```
int i; --> variable redeclaration
```

According to the 1978 Kernighan and Ritchie C standard (K&R C), C compilers will not generate an error or a warning if a function declaration and a call to that function have a different number of arguments. This can lead to bugs which are extremely difficult to find. To avoid these problems, SCIL_Image checks on the number of arguments in function declarations and function calls, and generates an error message when they differ. As an example, the command **strcmp** - which calls the function `strcmp()` - expects two arguments:

```
i = strcmp("Just one argument");
```

```
i=strcmp("Just one argument");--> more function arguments expected
```

```
i = strcmp("Here are", "three", "arguments");
```

```
i=strcmp("Here are", "three", ""arguments");--> too many function arguments
```

A K&R C compiler would not have complained in this case. SCIL_Image thus checks syntax more strictly than you may be used to.

When you explicitly test for errors (for instance in opening a file), it is common practice in C programming to leave the program with `exit()` upon an error. However, you should be careful doing this, because it will actually *quit SCIL_Image*, rather than just the program, which may not be what you want. The reason is that `exit()` is a standard C function, known to SCIL_Image, which interprets it as a command to quit. You will get the functionality of `exit()` by using `return()` in a `main()`, or by using `S_error(0)`.

Even when a program is syntactically correct, it may still contain a programming error, only detectable on execution. Such program errors may originate from illegal memory references, illegal instructions, or floating point exceptions, to name a few. If one of these errors occurs, the SCIL_Image interpreter will *attempt* to stop execution, print the corresponding error message and return to the Direct Command Mode. For example, if an attempt is made to redirect a NULL valued pointer, execution of the following **for** loop will be stopped, enabling the user to correct the mistake:

```
int *ptr;
for( i = 0; i < 100; i++ ) printf("%d ", *ptr++ = i);
for( i = 0; i < 100; i++ ) printf("%d ", *ptr++ ==> indirection of a NULL valued
pointer;
```

The other program errors generate similar error messages. *However, if your error is sufficiently severe, it may crash SCIL_Image. This will delete all changes you have not saved, and destroy all unsaved images.* You may even have to reboot. If you are working in programs you do not trust, it is always wise to often 'Save' files and relevant images. Also, use **logon** (see "Program Development Commands", page 4-9) to keep a record of the commands you gave.

Features of SCIL_Image's C-interpreter

In SCIL_Image version 1.4, the C-interpreter does *not* have the full functionality of the Kernighan and Ritchie 1978 C-language. A small number of topics are missing, some of which may become available in future versions of SCIL_Image. Furthermore a small number of ANSI-C features have been added or are recognized (but further ignored) to improve flexibility.

The main differences are:

- ANSI-C calling convention, arguments are no longer promoted to **int** or **double** before they are passed to a compiled function.
- no initialization of arrays and structures
- no goto statement and labels
- no bitfields
- octal bit pattern setting is only possible for single characters, not in strings
- no comma operator is supported, except in a for-loop. (As in: **for(i=1, j=1; i < 2; i++)**)
- the ANSI-C preprocessor construction **#if defined(.....)** is supported.
- the ANSI-C preprocessor keyword **#pragma** is recognized and completely ignored.
- In the interpreter the preprocessor symbol **INTERPRETED** is defined.

The differences in scope rules are:

- Extern only works for interpreted variables, compiled variables can only be referenced by using the VAR keyword in the comfiles and rebuilding SCIL_Image (see "Making a Command Description File (CDF)").
- Static variables are NOT supported. In older versions of SCIL_Image (up to version 1.2) a **static** variable was a global variable in the interpreter that could not be removed during a SCIL_Image session. Now the keyword **permanent** is available for this type of variables.
- The ANSI-C keyword **const** is recognized but not implemented, the keyword is ignored.

Chapter 5 Advanced SCIL_Image

This chapter describes the advanced use of SCIL_Image. It shows how to add new commands to the system and how to change the user interface. This section is rather technical - more general topics concerning SCIL_Image, such as the Direct Command Mode and the Program Mode can be found in Chapter 4.

Read this chapter if:

- You want to add new commands to the system.
- You want to add help information to the on-line manual.
- You want to change the user interface.

Do not read this chapter if:

- You have not already read Chapters 2 and 3 (and preferably also 4).

Adding New Functions to SCIL_Image

This SCIL_Image version contains many basic image processing programs. For your own applications, you probably need special commands, which may or may not be composed using those basic programs. Very likely, you will want to build up your own library of programs that are useful for your applications. SCIL_Image allows you to do this in two ways. One is to *extend* SCIL_Image with *interpreted programs* as we discussed in Chapter 4. The second is to create a completely *new version* which includes your own *compiled functions*. You will be able to reach those functions from menus, or in the Direct Command Mode - they have thus become actual SCIL_Image *commands*. Also, you can use them as building-blocks for more advanced functions.

There are three files which describe new functions or programs to SCIL_Image (apart from your own files that define what they actually do). They are: 'scilimage14\comfile', 'scilimage14\prog\sysfunc.c' and 'scilimage14\prog\overload.c'. You may not need to generate all three files, depending on what kind of functions you want to add.

- **comfile**

If you want to extend SCIL_Image with your own *interpreted programs*, all you need to do is generate a new 'scilimage14\comfile'. This file will contain information on the menus, functions, default values and dialog boxes for all commands in the standard SCIL_Image version and for your interpreted additions. You supply this information through a *Command Description File* (CDF for short), recognizable by the suffix '.cdf'. (In **Session Three: Using the SCIL_Image Command Line Mode**", page 3-21, this file was called 'personal.cdf'.) We give details of the CDF file in **Making a Command Description File (CDF)**" on page 5-3.

- **sysfunc.c**

If you want to make a new version of SCIL_Image that contains your own *compiled functions* as new *commands* in SCIL_Image, you also have to make a file 'scilimage14\prog\sysfunc.c', which contains references to all functions, including your own. This file is generated by **mksysfnc**' (started by the **nmake** utility). You have to compile and link it with SCIL_Image. Details are given in **"Creating a New Compiled SCIL_Image Version"** on page 5-16.

- **overload.c**

If you want to make a command that accepts arguments of more than one type (for instance, image processing commands that work both on integer-valued and on float-valued images), you have to make the file 'overload.c' which contains this overload information. You do this using the program 'mkoverld' (started by the **make** utility). You have to compile and link it with SCIL_Image. Details are given "**Creating a New Compiled SCIL_Image Version**" on page 5-16.

Thus if you want to add an *interpreted* program to SCIL_Image, you only need to generate 'comfile'. This requires the actions we specified in the sample session **Session Four: Programming in SCIL_Image**" on page 3-26. The actions required to add a *compiled* function to SCIL_Image require the use of a C-compiler. They are explained in **Creating a New Compiled SCIL_Image Version**" on page 5-16.

Making a Command Description File (CDF)

One of the things you must do when adding new commands is create the Command Description File. In this section we describe its syntax, and how you can express your functional desires in it. For easy reference, let us repeat the file 'myfunc.cdf' from Chapter 3:

```

$MyMenu      $SCILIMAGE

FUNC myfunc
MENU MyMenu Options
OPTIONS NOT_COMPILED
ARGS
    filename - trui - * My File
    image - A - - My Image
    odd - 127 1 255 Only odd values
#
    
```

Before we begin describing the syntax, we have a warning.

IMPORTANT: MAINTAIN BACKUPS !

Please take extreme care when you make changes to your CDF. If you make a mistake in your CDF, this mistake propagates into the 'comfile', and SCIL_Image will not start at all. Be sure to keep a backup of the last working version, so you can restore the old version if needed.

The following sections are written in a formal syntax to prevent ambiguities. The syntax of the Command Description File is:

```

CDF ::= <CDF_entry>*

<CDF_entry> ::= <comment entry>|
                <menu entry>|
                <translate entry>|
                <variable entry>|
                <command entry>
    
```

A CDF may consist of one or more CDF entries (the * means 'one or more of the previous'). A CDF entry can either be a *comment* entry, or a *menu* entry, or a *translate* entry, or a *variable* entry, or a *command* entry.

comment entry

A **<comment entry>** is indicated by a hash sign # at the beginning of a line:

```

<comment entry> ::= # <any text>
    
```

menu entry

The **<menu entry>** is used to specify the way the menu looks, and the placement of your commands in it. The syntax of a menu entry is:

```

<menu entry> ::=
    $<menu_name>    [ $<parent menu_name>* [comment] ]
    
```

\$<menu name>

A menu entry starts with a dollar sign (\$) followed by the name of the new menu.

[\$<parent menu>*]

Optionally, the new menu name is followed by a list of parent menus. A parent menu is referenced by a dollar sign (\$) followed by the name of the parent menu. The new menu will be added as a child (sub-menu) to each of the parent menus. All parent menus referred to in the description must have been specified before.

[comment]

Any text after the list of parents will be considered as a comment for your own use, and will not be interpreted by the system.

translate entry

the **<translate entry>** contains C-type definitions for the CDF types used in function argument specifications. For each of the argument-types of commands described in the CDF files, SCIL_Image needs to know the exact C-type to be able to call the function correctly.

The syntax for the type definition is:

```
<translate entry> ::=
TRANSLATE      cdf-type      "C-type"
```

TRANSLATE cdf-type "C-type"

A translate entry starts with the key word **TRANSLATE** .

The **cdf-type** is an argument-type (that is not also a C-type) used to describe the types of the command parameters in the ARGV section of a **<command entry>**. A complete listing can be found in "**command entry**" on page 5-6.

The **"C-type"** is the basic C-type of the corresponding **cdf-type**. E.g. an "odd" is represented in the interpreter as an `int`, the TRANSLATE line in the CDF file therefore reads:

```
TRANSLATE    odd    "int"
```

The translate entry must be located in the CDF file before the cdf-type it describes is actually used in the description of a command..

variable entry

The **<variable entry>** contains global variable descriptions, one for each global **compiled** variable. Such a variable description is used by SCIL_Image to access these variables from within the interpreter. The syntax for a variable description is:

```
<variable entry> ::=
VAR  name  ["type"]
```

VAR name ["type"]

A variable entry starts with the key word **VAR** followed by the **name** of the global (compiled) variable.

The **type** of the variable may be specified (surrounded by string delimiters), but this is not required. If no type is specified the variable is assumed to be an `int`. Valid types are the standard C types and pointers to them. Structures and unions must be completely defined in the **type** string. It is not possible to refer to defined structures like `FILE` or `IMAGE`.

command entry

The **<command entry>** contains command descriptions, one for each command. Such a command description is used by SCIL_Image to build a dialog box for the command, and to check arguments, defaults and the expansion of names. The syntax for a command description is:

```

<command entry> ::=
    FUNC name ["type"] [alias]
    [MENU      [$]menu]* [menu-options]
    [MANUAL    name]
    [OPTIONS   {NOT_COMPILED | NO_EXPAND}]
    {ARGS [type* [, ...]]
    <tab>[type vise default min max prompt]* }
    
```

FUNC name ["type"] [alias]

OBLIGATORY

A command entry starts with the key word **FUNC** followed by the **name** of the function that defines the command.

The **type** of the function may be specified (surrounded by double quotes), but this is not required. Valid types are the standard C-types and pointers to them. In SCIL_Image it is not possible to refer to defined structures like `FILE` or `IMAGE`. If a function returns a pointer to such a structure, `char*` or `void*` should be filled in. If no return type is specified the command is assumed to return an `int`.

The **alias** field can be used if the name of the function, as it is used in SCIL_Image, differs from the *actual* name of the compiled function. This feature should be used rarely, if at all, since it is a potential source of confusion.

If you wish a menu item containing **blanks** (such as 'Save As'), then you should denote this by an @ (here: 'FUNC Save@As'). The actual function name which will be executed should then have an '_' at the blank (in this example, 'Save_As'). (Of course you could have had the same function called if you had specified the entry 'FUNC Save_As', but then the menu would have shown the underscore, which is less elegant.)

[MENU [\$]menu* [menu-options] *OPTIONAL*

If the command is to appear under one or several menus it must be registered with those menus. This is accomplished with the keyword **MENU** followed by one or more menu names. The menus must have been previously declared in the **<menu entry>**. (The \$ signs are optional and only for compatibility with older versions of SCIL_Image.)

The **menu-options** keywords follow the menu names. They are used to influence the layout of the menu and to change the behavior of the automatic generation of dialog boxes. The menu-options keywords are:

MENUSEP

A separator line is drawn directly beneath the item (in all the menus this item is present).

MENUKEY=key

key is a single character that activates the item in the current menu. In the menu the first **key** character of the item name is underlined.

SHORTCUT=key| [<modifier>Fnr]

key is a single character that combined with the shortcut key activates an item from anywhere in the menu-tree. [**<modifier>Fnr**] is a function key (with a modifier key (Shift, Ctrl or Alt) that activates an item from anywhere in the menu-tree. **nr** is the number of the function key. The shortcut-key is displayed next to item in the menu. Since the shortcut keys activate an item anywhere in the menu tree, they must be unique.

BOLD, ITALIC, UNDERLINE

These keyword change the text style of the menu item. These keyword may not be implemented on all platforms.

NOPRINT

The command is not printed in the (dialog) history window.

TOGGLE

The argument of the command toggles between 0 and 1 and the command is executed directly without generating a dialog-box. The start value depends off the argument depends on the optional CHECK keyword following the TOGGLE. Using this keyword on commands that have more than one argument results undefined behavior.

CHECK

The command argument is on by default. Used for the TOGGLE keyword. Using this keyword on commands that have more than one argument results in undefined behavior.

DISABLE

The command is disabled in the menu, it can not be chosen.

DIRECT

The command is executed immediately. (No dialog box is generated. To indicate this, no ellipses (...) are printed after the command name)

DIRECT...

The command is executed immediately. But because the command itself generated some kind of dialog-box, the ellipses are printed after the command name.

[MANUAL name]

OPTIONAL

The text which describes a function's syntax and behavior is normally stored in a file named after the function with the suffix '.man'. If this is the case, the **MANUAL** field does not need to be specified. However, for a related group of functions, it can be useful to group the explanations of those functions in one text file. Rather than copying the file several times, the **MANUAL** option can be used to identify the file which contains the explanation for the group. The file must be in one of the directories specified by the environmental variable `SCIL_MAN`.

On the **MS-Windows** platform, the manual pages of all functions are integrated in the help-file of `SCIL_Image`, the **MANUAL** keyword therefore is not used on this platform.

[OPTIONS {NOT_COMPILED | NO_EXPAND}] **OPTIONAL**

If the **OPTIONS** field is absent it is assumed that the described function has been compiled. If the function has not been compiled, then the keyword **NOT_COMPILED** must be chosen. `SCIL_Image` now knows that it should find the function defining the command in a UFO file 'name.c' (where 'name' is the name of the function, see "The Programming Mode: Interpreted C-functions and UFOs").

Typically, SCIL_Image expands any command you may give it, which means that it fills in default values of the non-specified parameters. This may be undesirable, for instance if the command can have a variable number of arguments. For such commands, you should specify the option **NO_EXPAND**.

ARGS [ctype*[,...]] **OBLIGATORY**
<tab>[type vis default min max prompt]*

The **ARGS** field followed by zero or more lines beginning with a **<tab>** character indicates the beginning of the description of the function's arguments. This field is obligatory.

If, and only if, **OPTIONS NO_EXPAND** is used (see above), then you need to specify the arguments by listing them in an ANSI-C prototype-style, e.g. for the "printf" function:

ARGS char *, ... ("..." specifies a variable argument list)

In all other cases *each* argument must be described, as follows:

<tab> The argument list for a function continues until the first line which does not start with a space or tab. A **<tab>** is often used to improve readability. Any line starting with white space is assumed to contain an argument description.

Therefore you should not have any empty lines at the end of your CDF file.

type The **type** field specifies the argument type. The admissible types are in two categories, *C variables* and the *SCIL_Image special types*.

The C variables are:

int, uint
short, ushort
long, ulong
float, double
char, uchar

(**uint** stands for **unsigned int**, but since only a single word can be used in the CDF, it has to be abbreviated - **ushort, ulong** and **uchar** are similar.)

The SCIL special types are:

text pass argument unaltered
string text (surrounded by optional double quotes)
odd odd integer value only
even even integer value only
confirm Yes, No, 1, 0

switch	On, Off, 1, 0
choice	one choice out of a list of choices
toggle	multiple choices out of a list of choices
filename	ask for name of a file
filesave	ask for file name by popping up savebox

A more detailed description of the SCIL special types is given in "SCIL_Image Special Types", page 5-10.

vis This field is reserved, preferably it should contain a minus sign.

default This field holds the default value of the argument. The default may be a string or a SCIL variable (for a SCIL variable, the current value is substituted). When a command is issued with too few arguments, defaults are used for the missing parameters. The default values are also presented in the dialog box.

min, max The interpretation of the **min** and **max** fields depends on the argument type. For numerical types, the value of a given function argument is checked against the legal value range specified in the **min** and **max** fields. If a value is outside the range SCIL_Image will complain with the message "*out of range*". The user will then be prompted to enter a value within the legal range. An isolated minus sign may be entered in the **min** or **max** field - in that case SCIL_Image does not check your entries. Many of the SCIL special types use the **min** and **max** fields for special purposes. This is described in in "SCIL_Image Special Types", page 5-10.

prompt The **prompt** is used to label fields in the dialogue box: it is the text that will specify what needs to be entered. It also appears on the screen if a question-mark has been entered or if an invalid value has been entered (in that case it includes the display of the default within brackets '[]'). If the user enters too few parameters followed by a question-mark, the prompting will appear for each of the missing parameters.

SCIL_Image Special Types

We saw in the previous section that there are some function types that are not part of standard C, but particular to SCIL_Image. We discuss these now in detail, as well as their roles in the CDF. Some of the argument types make special use of the **min** and **max** fields, and others do not use those fields at all. Whenever a field is not used, a minus sign should be filled in.

text, string **text** differs from the type **string**. A **string** is a piece of text, a **text** need not be a piece of text, it may also contain SCIL variables or values . The **min** and **max** fields are ignored for this argument type, and should contain a minus sign. An example:

```
# scil.cdf
#
FUNC time      scilcommand
ARGS
    text  in    display_image -    -    Command
#
```

confirm is used for variables that indicate a Yes/No choice. The **min** and **max** fields are ignored. An example:

```
# image.cdf
#
FUNC set_display_mode
MANUAL st_dmode
MENU Display
ARGS
    image in A - - Image
    choice in - "NORMAL{0}|LIN_STRETCH{4}|LOG_STRETCH{8}
    confirm in Yes - - Global for 3-D Images
    switch in Yes Yes No Direct display
#
```

switch can be used for arguments that can have two values (not necessarily Yes or No). Internally, these values are represented as 1 and 0, but you may give them any name: the **min** field contains text which is used as a symbol for '1', and the **max** field contains text which is used as a symbol for '0'. An example:

```
# image.cdf
#
FUNC sobel_diff
MANUAL sobe_dif
MENU Filter
ARGS
    image in A - - Input Image
    image out B - - Output Image
    switch con sum sqrt sum Sum OR Sqrt of quad
#
```

choice is used for variables that select among a number of pre-defined choices. Each choice can have an argument value which is passed to the function when the choice is selected. Often this is an image identifier or an integer. The choices are listed, separated by bars (|) in the **min** field. The **max** field is empty and should therefore contain a minus sign. An example:

```
# im_aio.cdf
#
FUNC aio_label
MANUAL aio_labl
MENU Single_Objects Conversion
ARGS
    image in      B      -      -      Input Image
    image out     C      -      -      Output Image
    choice con    8      "4|8" -      Connectivity
#
```

A **choice** may have an *alias* associated with it. In this way, SCIL_Image can prompt you to select among meaningful terms rather than among integers. Suppose for example, the valid choices for a particular argument are 1, 2, and 3, but that they actually are choices of the X, Y or Z axis. You can make this explicit by substituting "1|2|3" in the **min** field with "X{1}|Y{2}|Z{3}". In the dialog box, and at the command line, you will be offered a choice among X, Y, and Z, rather than among 1, 2, and 3. An example:

```
# image.cdf
#
FUNC contour
MENU Special_Points
ARGS
    image in      B      -      -      Input Binary Image
    image out     B      -      -      Output Binary Image
    switch in     Off    On      Off    Set Edge Pixels
    choice in     8      "4|8" -      Connectivity
    choice in     Object "Object{1}|Background{0}" - Make Contour of
#
```

(In this example, note that because there are only two valid choices, with values 1 and 0, a **switch** could have been used rather than a **choice**, with the same flexibility in defining the terms displayed in the dialog box. In this case, it is primarily a matter of preference whether a **switch** or a **choice** is used as the argument type.)

toggle resembles the **choice** type in many ways. The difference between the two is that with a **toggle** you may choose more than one selection from the given list, whereas the **choice** type allows only one selection. The syntax for the **toggle** type is exactly the same as the **choice** type. SCIL_Image performs a logical OR on the choices and calls the function. As with choice, you can attach aliases for easier readability. An example:

```

# im_aio.cdf
#
FUNC measure "void *"
MENU Single_Objects
ARGS
    image in      A      -      -      Grey image
    image in      B      -      -      Binary image
    int           con    0      0      262144 Garbage level
    confirm       con    No     -      -      Interaction
    toggle con    AREA|PERI  "AREA{1L}|PERI{2L}|CR{4L}|BE
    toggle con    AREA|PERI  "AREA{1L}|PERI{2L}|CR{4L}|BEND{32768L}|XMIN{
8L}|XMAX{16L}|YMIN{32L}|YMAX{64L}|WIDTH{128L}|HEIGHT{256L}|GRAVX{512L}|GRAVY{1
024L}|ANGLE{2048L}" - Shape
    toggle con    GREYVAL  "GREYVAL{1L}|TRANSMIS{2L}|OD{4L}" - Density
    confirm       con    Yes   -      -      Print results
    filesave     con    "-"   "."   "*"   Store in file
#

```

If you choose PERI and CR as choices for the shape argument, the appropriate measure function is called with the value 2 | 4 = 6.

filename is used when the argument is the name of a file. For this argument type, the **min** and **max** fields are used to supply the SCIL_Image file selector with necessary information. The **min** field is used to specify a default directory. The **max** field may contain several patterns to be matched in the file name. You can use this, for instance, to specify the suffixes of the filenames you want shown in the dialog box (like **"*.im"** if you only want to see image files). Only the files in the folder which match the patterns in the **max** field will be offered for selection. An example:

```

# image.cdf
#
FUNC readfile
MENU I/O
ARGS
    filename con  "trui" "."   "*.ics *.im *.dat *.tif*"  Filename
    image out  A      -      USE_NAME{0L} Image
    int       con  60   0      1000 X Position
    int       con  60   0      1000 Y Position
#

```

filesave is identical to **filename** on X-window systems. On other platforms the filename type may not always offer an editable text field to fill in a new name for a file, while a **filesave** type will.

```
# image.cdf
#
FUNC writefile
MANUAL writefil
MENU I/O I_O
ARGS
    image    in    A    -    -    Image
    filesave con "-"  "."  "*"  Filename
    choice con  ICS_F "ICS_F{1}|TIFF_F{2}|TCL_F{3}|JPEG_F{4}" - File
Format
#
```

the types **image**, **clut**, **v_object**, **histogram**

In several examples, you have seen the type **image** used. This is actually not a new type, but a different name for a type choice variable, with some additional properties. In the next session, we describe these derived types, and show how you can construct them yourself.

Very Advanced SCIL_Image: New Types

Whenever SCIL_Image encounters a type which is unknown, it calls a list of functions that try to interpret the type. You can add functions to this list. This gives you the opportunity *to define types of your own*, if you need something beyond the special types provided with SCIL_Image.

To add a *type handling function* you must use the standard function `set_extra_type_func()`. This function takes as its argument a pointer to a function.

```
int set_extra_type_func(int (*fun_ptr)())
```

The pointer is to a function, e.g. `my_type_func()`, which contains the behavior of your new type. The synopsis of this function is:

```
int my_type_func( char *name, char *type, char *deflt, char *minstr, char
*maxstr, char *prompt)
```

Internally, the first thing your function should do is to compare the input `type` to the name of the new type that it defines. If the given type is not for your function you should return the value 0. If the type is for your function, then you must change the given information into one of the know types, and return the value 1.

We wrote such a function to implement the **image** type as a variation of the **choice** type. It does the following:

- 1) If the `type` is not **image** then 0 is returned.
- 2) The `type` is changed to **choice**.
- 3) `minstr` (which is the variable corresponding to the **min** field) is changed to hold a list of all the available image names, with the image address as an alias. So if the images 'A', 'B', and 'scratch' are available, `minstr` will be changed to "A{234568}|B{236888}|scratch{238444}" (or something similar, depending on the image pointers).
- 4) `maxstr` (which is the variable corresponding to the **max** field) may be used to pass on extra information, such as a symbolic constant. Therefore, if the `maxstr` field is not equal to the minus (-) sign, the information in the field is added to the list with choices in `minstr`.
- 5) The value 1 is returned to indicate that our function has successfully transformed an unknown type into a known type, in this case **image** to **choice**.

Several types have been added in this way. These types are:

- | | |
|------------------|---|
| image | presents a list with available images. |
| im_type | presents a list of image types (used for instance in the 'Image:Conversion:convert' command). |
| v_object | presents a list with available 'var_objects' (see "Non image data (var_objects)", page 6-24). The min field can be used to specify the 'var_object' class. If this field is used only var_objects of the given class are accepted. In the max field extra choices can be given. |
| histogram | presents a list with available 'histogram' objects (see "Histogram objects" on page 6-26). The min field is discarded; the max field can be used to specify extra choices. |
| clut | presents a list with available color lookup tables (see "Display lookup table" on page 6-7). |

Creating a New Compiled SCIL_Image Version

The description of the CDF files permits you to add your own, non-compiled programs to SCIL_Image, even making them accessible from the menu. However, they will still be *interpreted*, and are therefore slow and not fully accessible to other programs or functions (or to other users!). You may want to make a new version of SCIL_Image that contains functions that are important to your applications in a *compiled* form.

Prepare (write, and possibly debug) the program(s) in C that you wish to install in SCIL_Image. Let us use as an example a series of functions to generate various types of random noise. This means that you have to prepare the following documents:

My_src	directory
noise.cdf	your command description file (CDF)
noise.ovl	your overload file
noise.h	this file contains your constants
noise.c	this file contains your functions

The rules for composing the 'noise.cdf' file are given in **Making a Command Description File (CDF)**", page 5-3. The rules for composing the 'noise.ovl' file are given in **Function overloading**", page 9-27. The rules for composing 'noise.c' and 'noise.h' are just those of standard C (see any C manual, but also beware of the slight deviations of SCIL_Image's C, see "Features of SCIL_Image's C-interpreter", page 4-16). You might want to study the example in "Making a New Compiled Version of SCIL_Image", page 3-30.

First an important note on compiled functions in SCIL_Image.

IMPORTANT: Special include file for I/O !

When adding compiled functions to SCIL_Image that use printf(), fprintf(), putc() etc. to print text to the Worksheet or getchar(), scanf() etc. to read text and characters from the keyboard, the include file 'scil_io.h' must be included in the source file.

Perform the following actions:

- 1) Open the Microsoft Developer Studio workspace "c:\scilimage14\prog\scilimag.dsw" (if not already open) and add the file add_one to the project "mylib" (right click "mylib" and choose "Add files to project" and then select the file noise.c from the directory you put it in.)

- 2) Open de file "c:\scilimage14\prog\comfiles.mak" (set the "Open as" to text in the dialog box) and at the end of the COMFILES list add:

c:\scilimage14\prog\My_src\noise.cdf

and save the file. (fill in the appropriate path name if you saved the file "noise.cdf" somewhere else).

- 3) Open de file "c:\scilimage14\prog\overload.mak" (set the "Open as" to text in the dialog box) and at the end of the OVERLOAD list add:

c:\scilimage14\prog\My_src\noise.ovl

and save the file. (fill in the appropriate path name if you saved the file "noise.ovl" somewhere else).

- 4) For both the "comfiles" and "overload" project, choose the "Build" command on the context menu (when clicking the project with the right mouse-button).
- 5) Then rebuild SCIL_Image by choosing the "Build" command on the "scilimage14" project.
- 6) Start SCIL_Image again. It now contains your function as you defined it, available under menus you specified in "noise.cdf".

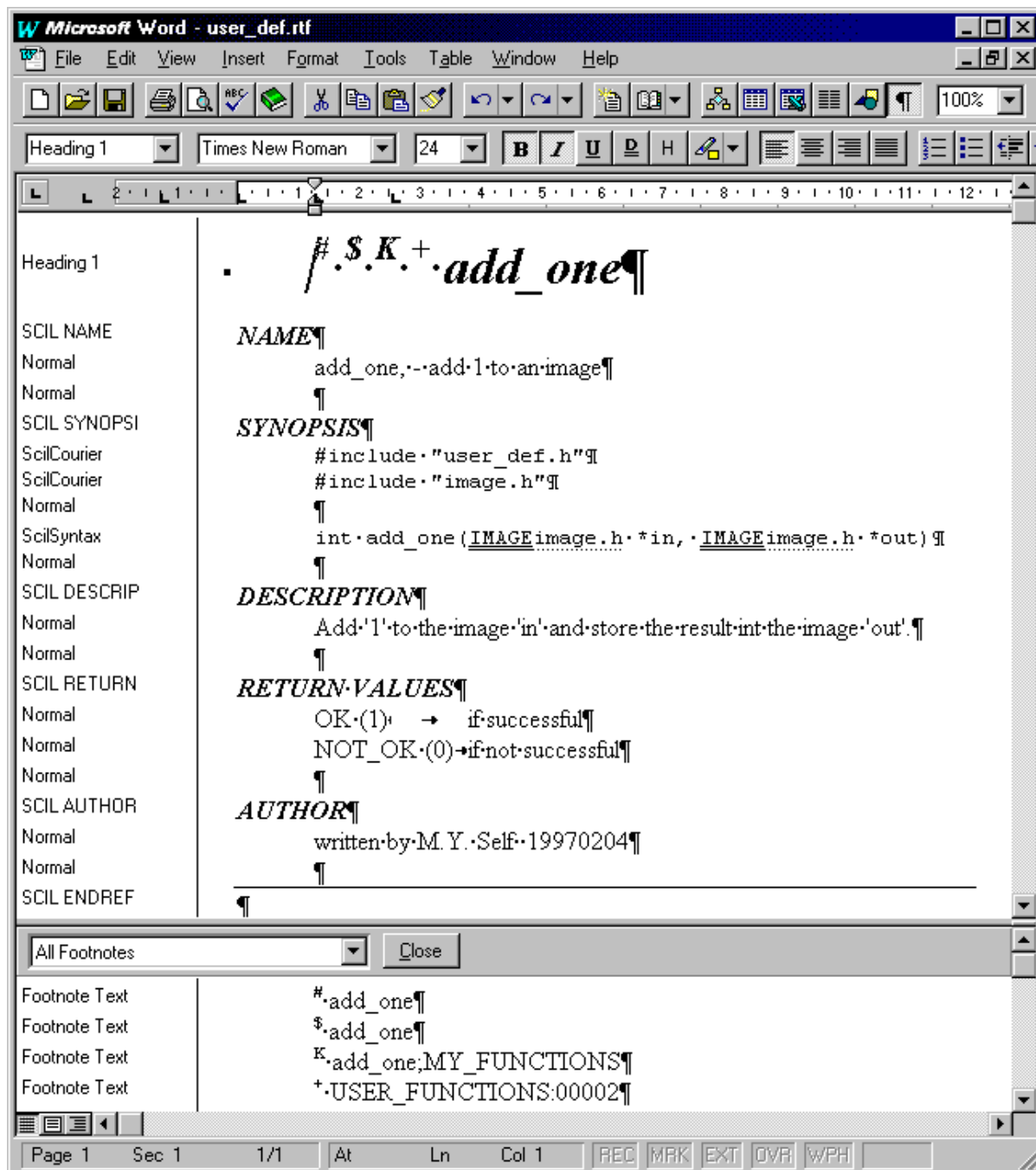
Please note that just must build the "comfiles" and "overload" projects by hand first because Developer Studio does not recompile the files "sysfunc.c" and "overload.c" when necessary.

Adding On-line Manual Files to SCIL_Image

Every function added to SCIL_Image may have an *on-line manual page*. This is a section in the file 'scilimage14\help\user_def.rtf', which gives the basic text, and defines links and keywords used to make it into a hypertext for the Help facility. You can make such a page yourself, and connect it to the standard SCIL_Image Help facility.

. You should do this by following the instructions on the Help page: '**User-defined Help Pages**'. It requires the editing, in Rich Text Format, of the file '**user_def.rtf**', located in the 'help' directory. This file contains a sample manual page that you can use to create your own manual pages. For this you need an editor that can read this format. After you are done

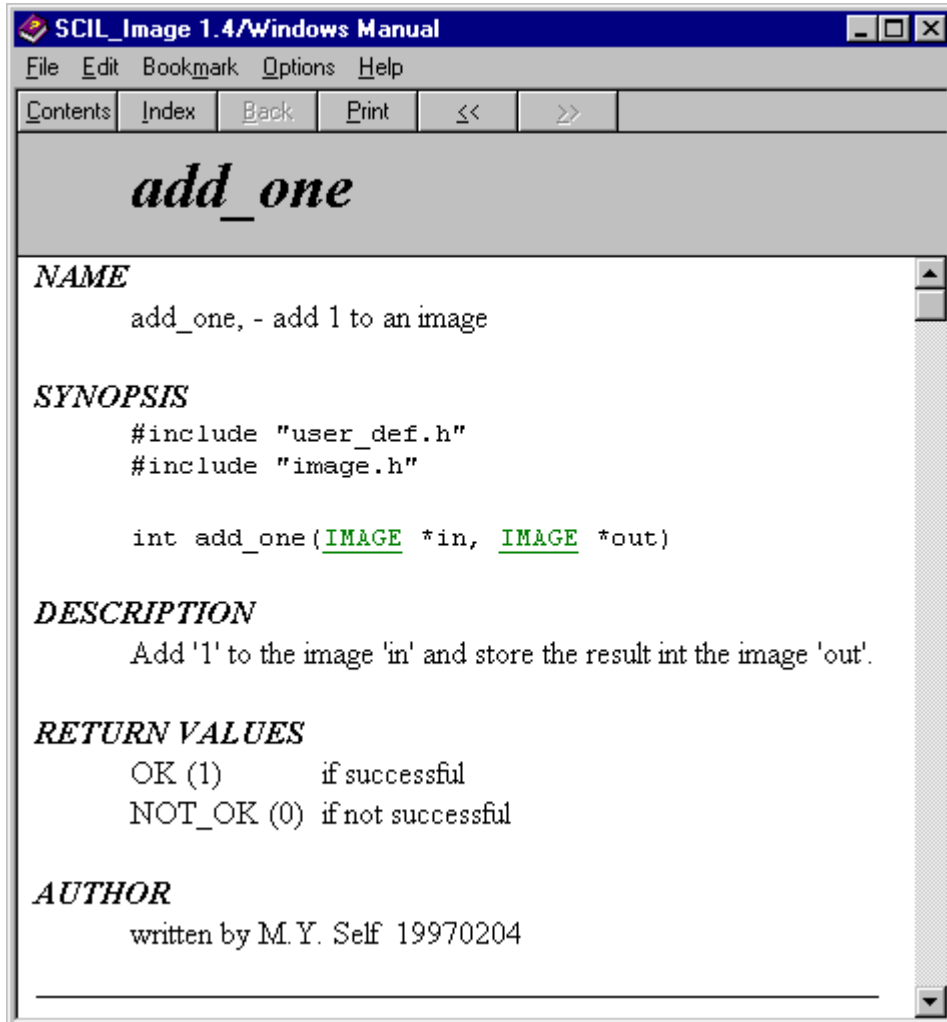
composing your Help page, according to the instructions, your editor window should look similar to this:



Save it, and start the "Help Workshop" from the "Microsoft Visual C++" environment. In the Help Workshop open the file 'scilimag.hpj' (from the help directory). To compile the help file:

Select 'File:Compile'

Compilation will take some time, the help compiler will minimize itself during compilation and restore itself to normal view after compilation. It generates the new '**scilimag.hlp**', which makes your Help page reachable from SCIL_Image like that of any other command:



As is customary in Windows' on-line help, the underlined colored terms represent links to other help pages, or pop-up windows (when the underlining is dashed).

Chapter 6 The Image 2.1 library in SCIL_Image

This chapter introduces some basic features of the Image 2.1 library as part of SCIL_Image.

Do not read this chapter if:

- You have not read "**Getting started**" (chapter 3). Specifically, it is assumed you know how to handle the menu & dialogue boxes and you know how to work with the command line editor.

Read this chapter if:

- You want to process images in Image.
- You want to know more about image types.
- You have basic knowledge of image processing.

Knowledge of the C language is not required for this chapter.

Introduction

In chapter 1, it is explained that the image processing functions all reside in the Image library which can be used independently from the SCIL_Image environment. In order to use Image effectively for image processing, it is necessary to know something about its infrastructure. This chapter describes the Image infrastructure and the SCIL_Image specific user interface on Image. Any individual image processing commands that are mentioned are documented in the reference manual(s)/help file.

One of the features of Image is the support of various image types. The image-types implemented in this release are:

<i>image type</i>	<i>name in Image</i>	<i>(in title bar)</i>
grey-valued 2-d	GREY_2D	(g2d)
binary bitmapped 2-d	BINARY_2D	(b2d)
floating point 2-d	FLOAT_2D	(f2d)
complex 2-d	COMPLEX_2D	(c2d)
color 2-d	COLOR_2D	(col2d)
labeled 2-d	LABEL_2D	(l2d)
grey-valued 3-d	GREY_3D	(g3d)
binary bitmapped 3-d	BINARY_3D	(b3d)
floating point 3-d	FLOAT_3D	(f3d)
complex 3-d	COMPLEX_3D	(c3d)
color 3-d	COLOR_3D	(col3d)
labeled 3-d	LABEL_3D	(l3d)

Table 6-1: image types available in Image

In Image, the user does not need to know the type or dimensions of an image an operation will result in. The package automatically sets the size and type of the output image based on the operation and the type and size of the input image. This adjustment also occurs when the output of an operation is directed to the input image.

At several points in the remainder of this chapter examples of commands are given. The user is encouraged to read this chapter while running SCIL_Image and try the examples. All examples are given as command lines. Most of them can also be issued using the menu, as described in Chapter 3.

Please note that in some examples, not all parameters are specified in the text, but are set to their default values by the command expander. The reader should refrain from specifying the missing parameters, because the resulting image type and contents of an operation may be assumed in subsequent examples

Image infrastructure

Invalid operations

In the course of processing an image, various errors may occur. For instance, the specified function arguments may be of the wrong type or out of range. A specified input image may be of an inappropriate type for the operation, or may not even exist. When errors of this type occur, SCIL_Image pops up an alert box to warn the user. Before continuing interactive work, the user is required to respond by clicking the 'Continue' or 'Stop' button in the alert box. To see how an alert box looks, type the following command lines:

```
readfile trui
threshold A B
threshold B C
```

The second **threshold** command request the system to perform the **threshold** operation on the binary image B. Because the **threshold** requires a grey valued input image, the system pops up an alert box with the text:

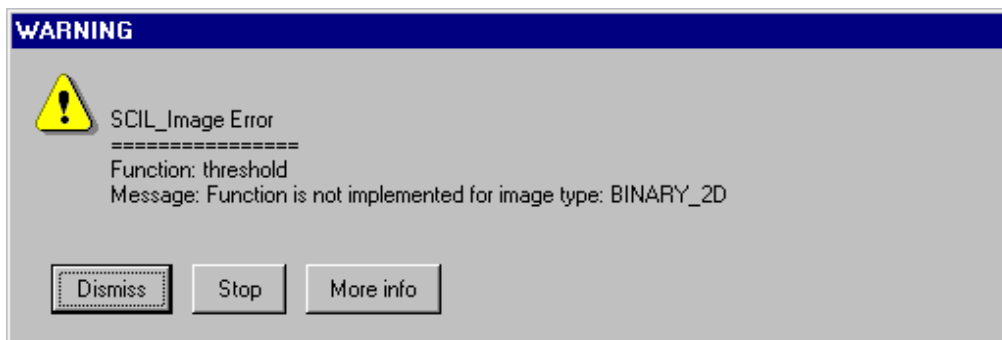


Figure 6-1: Alert box on incorrect image operation

The first line 'SCIL_Image Error' indicates that this error-message is generated by SCIL_Image. The next line 'Function: threshold' shows the name of the function that caused the error and the line starting with "Message: " describes the error. At the bottom of the box are three buttons. "[Dismiss]" removes the alert box and continues with the next statement (if any) in an interpreted program. "[Stop]" also removes the alert box and stops running the interpreted program (if one was running). In this example, only one command was given, so the difference between "Dismiss" and "Stop" is not noticeable. The "More info" button shows a new alert box that displays a "stack-trace" of the location of the error. That info is generally most of interest for those that write new image processing functions.

Image display and window management

An important part of interactive image processing is the display of the image. Several commands are available for this purpose, as can be seen in the 'Display' menu.

Please note when reading this section the distinction between an image and a display window. The dimensions of the image and the type of the data stored in the individual pixels depend on how the image was declared, or, in some cases the operation of which it is the output. The display window is a separate entity acting as a viewport on the data (with an independent size).

Also note that SCIL_Image runs on several operating systems. As a consequence the handling of the image displays (windows) can be vary between the different systems. In the text ,these differences are marked with: **UNIX**, **Macintosh** and **MS-Windows**. If you are in doubt about whether a certain operations is valid for the platform you are working on, please refer to the (on-line) manual page of that operation. When the operation is NOT supported on all systems, the **PLATFORM** section of the manual page will list the systems for which is valid.

Mouse buttons

On most **UNIX** systems the mouse has 3 buttons which each have a meaning in SCIL_Image. On the **Macintosh** however only, one button is available. To simulate the MIDDLE en RIGHT mouse button on the Macintosh the mouse button must be used together with the Option-key for the MIDDLE mouse button or the Command-key for the RIGHT mouse button. Although in general **MS-Windows** is used in combination with a two-button mouse, SCIL_Image for Windows supports all 3 buttons when using a three buttons mouse.

The title bar of image windows

In the title bar of the display window, information is displayed about the image. First, the name of the image is given. For instance, the names of the images supplied by SCIL_Image at start up are A, B, C and D. This is followed by a code in parenthesis, representing the type of the image (see Table 6-1 for the meaning of the code). Finally, the dimensions of the image are given. For A, B, C and D at startup, this is 256 * 256, because each of the images contain 256 rows of 256 pixels.

For a 3D image, a number indicating the plane (or slice) currently displayed, is displayed in parenthesis following the other information.

The left mouse button

If the left mouse button is pressed inside a display window, information is supplied about the position and value of the pixel pointed at. On **UNIX** and **Macintosh**, a small window pops up

containing the information, on **MS-Windows** the information is maintained only in the status bar. In **GREY_2D** images, the information is shown in the format:

(XXX, YYY) VVV

Where **XXX** is the x-position, **YYY** the y-position, and **VVV** is the value of the pixel at that position. If the mouse button remains pressed and the pointer is moved around in the image, the information in will updated accordingly.

The small information box can be positioned in the title bar permanently (not follow the cursor) by use of the command:

point_im_display_buf(0,0);

The default behavior can be restored again by:

point_im_display_buf(0,1);

In the occasion that the window containing the information is unwanted:

auto_point No

If this command is used, no window will pop up. To restore the data display:

auto_point Yes

Displaying the image

At the completion of each operation on an image, the output image is displayed. If automatic image display is unwanted, it can be disabled with the command:

auto_display off

From then on, output images will not be displayed. For example:

copy_im A C

Operations are considerably faster with the display off. To turn back on:

auto_display on
copy_im C D

The image (in D) is now displayed, but image C is still invisible. To display image C:

display_image C

Both image C and image D are now visible in the display windows. The **set_display_mode** command enhances the visual contrast of an image, without modifying the image data. Try:

set_display_mode C LIN_STRETCH

For some images the difference is marginal, as is seen in the displays of images A and C. The pixels in image A already covered nearly the entire range of values that the display can handle. So in this case there is little improvement in contrast.

Displaying 3D images

In SCIL_Image, 3D images are displayed one slice at the time. To see how this works, read in a 3D image

readfile chromo3d C

To display a different slice, there are three options. One is to click the right mouse button (**Macintosh** : hold down the Command-key while pressing the mouse button) as described below. Another is to use the command:

next_plane C Up

to display the slice below and

next_plane C Down

to show the one above. Finally the display slice can be set explicitly.

set_display_slice C 8

Note in the title bar that slice number 8 is now displayed.

The right mouse button

The right mouse button is currently only used for the manipulation of 3D image display. As stated above, 3D images are displayed one slice at the time. If the right mouse button is pressed when the pointer is in the upper third of the display window, then the next slice of the image will be displayed. For instance, if slice 0 was displayed, slice 1 will be displayed after pressing the right mouse button. If, when the pointer is in the middle third of the window the right mouse is pressed, then the same slice will remain displayed. Finally if while the pointer is in the bottom third of the display window the right mouse button is pressed, the previous slice will be displayed. For instance, if slice 5 was displayed, then slice 4 will be displayed after pressing the right mouse button. This behavior can be disabled with:

auto_plane No

and enabled with:

auto_plane Yes

Changing a window's size

In order to investigate particular image details, it may be useful to enlarge the window in which an image is displayed. Likewise, to save display space, it can be beneficial to display images in smaller windows. If a window is resized using the tools provided with the window system, SCIL_Image will scale the view of the image accordingly. The size of an image display window can also be modified using SCIL_Image. Note that in both cases, the dimensions of the underlying image are unchanged. Only the display is modified. The command **set_window_size** is used to change the size of the image display window:

```
set_window_size A 512 512
```

To set the window back to the size of the image the same command can be used with the real size of the image (shown in the title bar). However, the command:

```
natural_window_size A
```

sets the window to the size of the underlying image (in this case 256 by 256 pixels).

Changing a window's position

The window's position on the screen can be manipulated in the same way as the size of the window. That is, either by dragging the window or by using the SCIL_Image command **set_window_pos** can be issued. For example:

```
set_window_pos A 300 0
```

Display lookup table

Often, it is useful to modify the colors used for image display, without altering the data in the image. The color lookup table mechanism enables users to select colors for image display. Several standard color lookup tables are created when SCIL_Image is initialized. These tables can be attached to an image to set a group of colors to be for displaying that image. If no table is attached explicitly to an image, a normal grey scale for grey, float and complex images is used. Binary and labeled images do not need a lookup table, because the method for displaying them cannot be modified. The available lookup tables at start up are :

<i>name</i>	<i>contents</i>
EMPTY_LUT	all entries black
BLUE_LUT	a blue scale
GREEN_LUT	a green scale
CYAN_LUT	green and blue scale mixed
RED_LUT	a red scale
MAGENTA_LUT	red and blue scale mixed
YELLOW_LUT	red and green scale mixed
GREY_LUT	normal grey scale
LABEL_LUT	scale that simulates label display
MULTI_LUT	primary colors each in a specified bit
OVERLAY_LUT_1..8	specified bitplane red
FALSE_COLOR_LUT	false color table

Table 6-2: default color lookup tables

A lookup table can be attached to an image as follows:

```
set_clut A OVERLAY_LUT_8
```

The lookup table 'OVERLAY_LUT_8' is filled with a grey scale with lookup values ranging from 0...255. For each value for which the binary representation has the 8th bit set, the corresponding color is red. This results in all values that are over 128 being displayed as red.

To get the grey scale back again :

```
set_clut A GREY_LUT
```

Image management

Aside from manipulating the display of an image, it is sometimes necessary to create a new image or change the image type. For instance, to create a test image on which to try out a certain operation, the type and/or sizes of the image may have to be changed explicitly instead of letting the infrastructure handle it.

We pause here to remind the reader of the difference between an image and a display window. The data is stored in an image, which has a size and pixel value range depending on the type and declaration. The display window is a completely separate entity acting as a viewport on the data with an independent size.

Creating and destroying images

When starting SCIL_Image, four images are created. If more images are needed, they can be easily created. To create a new image with the name 'fifth_image', issue the command:

```
make_image fifth_image
```

An image name may be any set of ASCII characters, but may not contain white spaces. A new image has popped up on the screen carrying the name 'fifth_image', of type 'g2d' and with the dimensions 256 by 256. This image can now be used for processing. For instance:

```
invert_im A fifth_image
```

Destroying an image is accomplished with:

```
destroy_image fifth_image
```

Changing image sizes

Changing the size of the image will destroy its contents. To change the size of image A:

```
change_image_size A 128 128
```

Image A now has dimensions 128 by 128. Each pixel in A has been set to zero.

Changing image types

To change the type of image C:

```
set_im_type C GREY_2D
```

Image C is now a grey valued 2d image. All pixels in C have been set to zero.

Converting images into other types

It is sometimes useful to modify the image type without losing the image data (as occurs with **set_im_type**). The command **convert** is supplied specifically for this purpose

```
readfile trui A  
convert A B FLOAT_2D
```

The grey valued image data in image A has been converted to floating point data in image in B. **convert** is capable of changing any type of image data into any other type. Apart from **convert** there are a number of commands that also convert the type of the data in the image to another type, all of which can be found in the 'Image:Conversion' menu.

When a 3D image is converted to a 2D image, each pixel in the 2D image will contain the maximum value of the pixels in the corresponding locations in all slices. This results in a specific 2D interpretation of a 3D image:

```
readfile chromo3d A  
convert A B GREY_2D
```

Filling images

There are several ways to fill an image with data. For instance, a data file may be read from disk. The most common way is to specify the image as the target for an operation. It is often useful however, to set the image contents specifically. For instance:

```
set_int A 128
```

sets all pixels of image A to the value 128. many commands to fill an image can be found in the 'Image:Generation' menu, each of which generates a test image. For instance, to fill image B with a chessboard pattern:

```
ti_block B 16 255 0
```

The most powerful tool for image data manipulation is the command **eval**, an expression evaluator that handles both GREY_2D and FLOAT_2D images. To fill an image with a grey ramp issue the command:

```
eval "D=xx"
```

This is one of the simple things that can be done using **eval**. See "Expression evaluation on images (eval)" on page 6-19 for a complete description of **eval**.

Region of interest (ROI) processing

To successfully extract the information sought in an image, it is often necessary to perform a large number of operations. Because the time required to perform an operation depends heavily on the size of the image, it is useful to restrict the area operated upon to the region containing data which is of interest. If the region of interest (ROI) is significantly smaller than the complete image, the execution time can drop dramatically. For example, if in an image of $256 * 256$ (65536) the region of interest is $64 * 64$ (4096) big, the number of pixels to be processed is a factor $4 * 4$ (16) smaller. This means that the time required to perform an operation will be reduced by a factor of (approximately) sixteen. In SCIL_Image, a ROI in an image can be defined by specifying the location of the top-left corner and the size. The ROI then will be treated as if it were a normal image. For instance, the result of an operation on a ROI will result in an image with the same size as the ROI. However the sizes of the ROI itself CANNOT be adjusted if the ROI itself is specified as an output image. Consider the following:

```
readfile trui A  
roi_define roi1 A 30 50 0 128 192 1  
copy_im roi1 B
```

A region of interest has been defined that starts at $x=30$, $y=50$ with $width=128$ and $height=192$. The command **copy roi1 B** copied the part of image A defined as 'roi1' to image B. Image B therefore is adjusted to match the dimensions of the ROI rather than those of image A. Let us see what happens when 'roi1' is used as the output image for a command. Try the following:

threshold B roi1

As a result of this command, the image-type of the ROI must be changed to binary. However, the ROI is simply a geometrical part of image A, and therefore image A must be changed to a binary image to store the result of the operation in the ROI. Only the ROI contains the result of the operation, the rest of the image is set to zero.

A ROI need not be rectangular, but may be arbitrary shaped (see the on-line manual of **roi_define**).

Image types

Grey valued images (GREY_2D & GREY_3D)

Data representation of grey valued images

Grey valued images consist of pixels, each of which contain an integer value (1,2, ..). These pixels are represented in SCIL_Image by a in the C language defined datatype 'PIXEL'. This type is defined to be a 'short int', which means that on most computers 16 bits (= 2 bytes) per pixel are used. Therefore the value ranges from -32768 to +32767 (a total of 65536 values).

readfile house A

If the pixels are examined (with the use of the left mouse button) it can be seen that every pixel has an integer value within the range [0,255]. This is because the camera this image was made with only used 8 bits per pixel. This type of camera is very common, and therefore most grey valued images encountered use only 8 bits per pixel. Why use 16 bits per pixel? There are devices that create images with more than 8 bits per pixel. There are also many operations on images with 8 bits per pixel which result in images than use more than 8 bits per pixel.

Usage of grey valued images

The grey valued image is the most common type used in image processing because its data representation matches the computer's data representation. The computer can therefore access the data quickly and easily. The amount of information that can be stored per pixel is sufficient for the majority of purposes.

Examples of grey valued operations

It is frequently useful to enhance the contrast in a grey valued image. We illustrate two operations available in SCIL_Image for doing so:

```
readfile maan A  
show_histogram A  
contrast_stretch A B 2 98  
show_histogram B  
equalize A C  
show_histogram C
```

The histograms for all three images were created in separate windows that are now position on top of each other, move the them around so you can compare them. To remove these histogram windows, **UNIX**: click the middle mouse button in the histogram window (**Mac**: Option-Click, **MS-Windows**: click the close box).

It is also often necessary to reduce the noise in a grey valued image. Several types of filters are available for this purpose:

```
readfile bnoise A  
percentile A B 3 3 4  
uniform A C
```

Binary bitmapped images (BINARY_2D & BINARY_3D)

Data representation of binary images

Each pixel in a binary images has value 0 or 1. One bit is therefore sufficient for the storage of each pixel. In SCIL_Image we store eight binary pixels in each byte so that a binary image requires only a fraction of the space occupied by a grey valued image. A border of long words is reserved around each binary image to enable very fast morphological operations (for more precise information see Chapter 11). The binary images in SCIL_Image are displayed in black and red on a color screen and in black and white on a monochrome screen, where black is

used for pixels of value 0 and red (white) is used for pixels of value 1. To create a binary image in B :

```
readfile bnoise A  
threshold A B 128
```

In window B, a red & black image has appeared. Pointing in it (with the left mouse button pressed) shows that all pixels in the image have value 1 or 0. The title bar of the display window confirms that this is a binary image (b2d). **threshold** mapped all pixels in image A with value less than 128 to '0' pixels in image B, and pixels that had a value of 128 or more to '1' pixels.

The colors in which the binary images are displayed can be changed by using the **bin_disp_colors** command:

```
bin_disp_colors WHITE BLACK  
display_image B
```

Usage of binary images

Because a binary image is made up of pixels, each of which is restricted to one of two values, the only information contained in a binary image is limited to which pixels are associated with an object and which are associated with the background. Binary images therefore play an important role in the investigation of object shape and form.

Examples of binary operations

Noise can be filtered from binary images as follows:

```
percentile B C 3 3 4
```

To extract object skeletons in an image:

```
readfile cermet A  
threshold A B 128  
invert_im B B  
hild_skelet B C
```

Often, only objects with a pre-defined minimum size are of interest. Removing smaller object can be accomplished as follows:

```
readfile cermet A  
threshold A B 128  
invert_im B B
```

```
erosion3x3 B C 8  
propagation C B D 30 8
```

By eroding several times, no 'seed' is left by small objects for the **propagation** operation that enlarges the eroded objects to their original sizes.

Floating point images (FLOAT_2D & FLOAT_3D)

Data representation of float images

Floating point images consist of pixels each of which can have any real value within precision limitations. Each pixel is represented by the C type 'float'. Though floating point images can contain more information than grey valued images, they are far less common. This is because computers used to work far more efficiently with integers, and because 'float' usually requires double the amount of space (32 bits per pixel). Try:

```
readfile trui A  
eval "B=0.005 * A"
```

Image B has now become a float image as is confirmed in its title bar. The image however, is no longer visible due to the low values of the pixels. By pressing the left mouse button in image B, the value of the pixels can be examined:

```
[(XXX, YYY) VV.VVV
```

Note that the value which pops up for each pixel is a real number rather than an integer.

Usage of float images

Floating point images are used in situations where accuracy is more important than speed, in particular when there are many mathematical operations on an image.

Examples of operations on float images

The trigonometric functions in SCIL_Image all operate on and result in float images. The following function calculates the sine for each point in image B:

```
sin_im B C
```

The result is almost invisible because all pixels have a low value. In order to boost the contrast of the image all values can be multiplied with a certain factor. To estimate that factor the maximum value that is present in image C has to be determined. This can be done by:

pix_maxval C

In the command window a line is printed that states:

```
maximum : 0.934
```

Now image C can be stretched by multiplying all values by 255 (to stay within the range 0..255):

```
eval "d=c*255"
```

Complex images (COMPLEX_2D & COMPLEX_3D)

Data representation of complex images

Each pixel in a complex image is a complex number and therefore has a real and imaginary part. In SCIL_Image, each of the parts is represented with the C type 'float'. Try the following:

```
readfile trui A
eval "B=0.333 * A"
complex_im A B C
```

Image C now contains a complex image, each pixel of which has its real part from the corresponding pixel in image A, and its imaginary part from the corresponding pixel in image B. Examining the pixels in image C (with the left mouse button) results in the following text:

```
"(XXX, YYY) Re: VV.VVV; Im: VV.VVV"
```

The first value after the parenthesis is the real part of the pixel and the second value is the imaginary part.

Usage of complex images

This type of image is specifically required for the Fast Fourier Transforms (FFT).

Examples of operations on complex images

To calculate the FFT:

```
fast_fourier C C Forward
```

Basic arithmetic operations such as adding and multiplying can be applied to complex images, or the real and imaginary parts can be examined and manipulated separately by

storing them in different images. They can then be rejoined in one image and further manipulated:

```
real_im C B  
mul_float B 1.34 B  
complex_im B C D  
fast_fourier D D Reverse
```

Labeled images (LABEL_2D & LABEL_3D)

Data representation of labeled images

Like a grey valued image, a labeled image is made up of integers with the C defined datatype 'PIXEL'. The interpretation of the values in a labeled image is however entirely different.

Usage of labeled images

The information in a labeled image is similar to that in a binary image. The pixels in the image either belong to an object or to the background. In a labeled image, however, each object in the image has its own number or 'label' used to distinguish it from other object in the image. The pixels in an image which have been determined to be part of an object, and are connected, are assigned a unique identifier. Each pixel in a labeled image has a value identifying it as a member of a set or object. Labeled images allow measurements such as size and shape to be performed on the image objects.

Examples of operations on labeled images

A labeled image can be created by labeling a binary image:

```
readfile cermet A  
threshold A B 128  
invert_im B B  
aio_label B C
```

Image C is now a labeled image (see the title bar). When a labeled image is displayed in SCIL_Image, different objects are displayed using different colors. Since only eight colors are used for displaying labeled images, the same color will be used more than once. However, separate objects of the same color have a distinct object id (check with the left mouse button). To show what kind of measurements can be performed on labeled images try the operation **measure**. The command has to be selected through the menu for clearness.

select 'Single_Objects:measure'
choose Interaction 'Yes'
hit 'DO IT'

A new image, called 'labelled_image' pops up in the middle of the screen which contains the same data as image C. To see this remove the dialogue box:

hit 'CANCEL'

Press the left mouse button while pointing at an object in 'labelled_image', several numbers will be printed in the command window. These numbers are the result of several measurements performed on the object. Stop the measurement by placing the pointer in the display window or the command window and type:

<RETURN>

Color images (COLOR_2D & COLOR_3D)

In this version of SCIL_Image (version 1.4), color images can be represented by one of five different color-models. The present models are RGB, (CIE) XYZ, (CIE) Lab, HSI and CMYK.

Data representation of color images

The color models of color images are each represented in a different manner:

RGB : each pixel is made up of a **R**ed, **G**reen and **B**lue component, each component (channel) occupies 1 byte in memory (C type : unsigned char). The three channels together with a fourth (extra and empty) byte are stored in one C structure.

CIE XYZ : each pixel consist of **three** floating point values, **X**, **Y** and **Z**, that are stored in one C structure. Each channel occupies 4 bytes of memory (C type : float).

CIE Lab : each pixel consist of **three** floating point values, **L**ightness, **a** and **b**, that are stored in one C structure. Each channel occupies 4 bytes of memory (C type : float).

HSI :each pixel consist of **three** floating point values, **H**ue, **S**aturation and **I**ntensity, that are stored in one C structure. Each channel occupies 4 bytes of memory (C type : float).

CMYK : each pixel consist of **four** floating point values, **C**yan, **M**agenta, **Y**ellow and **K** (sometime also named **B**lackness), that are stored in one C structure. Each channel occupies 4 bytes of memory (C type : float).

One of the reasons for the existence of different color-models (we implemented only a few of the many that are defined) is the suitability of each model for a specific task. The capability to convert one model into another model is therefore essential. Try:

```
readfile flamingo A  
convert_cmodel A B HSI_T
```

The color-model of a color image is printed also in the title-bar of the display window. When comparing image A and B, see that image A reads "col2d-RGB" (RGB model) while image B reads "col2d-HSI" (HSI-model). Also the left mouse button positioned in image A and B will display the difference between the images :

```
(XXX, YYY) R: redvalue G: greenvalue B: bluevalue
```

Compare this with the "same" image in B:

```
(XXX, YYY) H: h.hhhh S: s.ssss I: i.iiii
```

When processing color-images, an often used method that produces good results is to process each of the channel (or just 1 or 2) separately. To do so in SCIL_Image, the channels must be stored in separate images and then processed using a grey-value operation (either in integer or float). For instance reducing the intensity of the image in B:

```
set_im_type C float_2d  
copy_channel B C 2 0  
eval "c=c*.5"  
copy_im B D  
copy_channel C D 0 2
```

Usage of color images

At first glance, it may seem that recording and processing color images is a nuisance as they require a more sophisticated camera and a threefold amount of data. In many cases, however, using a color image simplifies the image processing task. Rather than having only 256 different grey values at one's disposal for every pixel, the amount for a color image is much larger, making it easier to group pixels on the basis of their pixel values only. Color data processing with the color data structure, gives the user access to one multi-valued digital color image, rather than keeping track of image data scattered over several single valued images, one for each channel of the color-model

Expression evaluation on images (eval)

A number of times in this chapter, the command **eval** has been used. **Eval** is an expression evaluator for images. This means that an operation on an image can be given in the form of an expression that is then executed on each pixel in the image. For instance:

```
eval "A=100"
```

sets all pixels of image A to the value 100. Let us try an operation on a real image:

```
readfile house A  
eval "B=A*2-80"
```

The value of each pixel in B is determined by multiplying the corresponding pixel in A by 2 and subtracting 80.

Eval can handle both GREY_2D, GREY_3D, FLOAT_3D and FLOAT_2D images. The type of image the result is stored in depends on the expression, the command given above all result in GREY_2D images but:

```
eval "C=123.45"
```

results in image C becoming a FLOAT_2D image. If the result of the expression is an integer then the result will be a GREY_2D image (or GREY_3D if the images used are 3D images), likewise if the result is a floating point number then the image will be a FLOAT_2D image (or FLOAT_3D).

The expression can be very complex because **eval** can handle all C-language operators as they are listed here with their priority :

<i>operator</i>	<i>associativity</i>	<i>priority</i>
() []	left	HIGHEST
- ! ~	right	
* / %	left	
+ - &+ &-	left	
>> <<	left	
> < <= >=	left	
== !=	left	
&	left	
^	left	
	left	
&&	left	
	left	
?:	right	LOWEST

Table 6-3: operator priority in eval()

The '&+' is a maximum operator and the '&-' is a minimum operator. Operators on the same line in the table have equal priority. Left-associativity means that if multiple operators of equal priority are used without brackets then the leftmost has the highest priority.

The operators '[' and ']' can be used for neighborhood operations, which means that not only one pixel can be referred to but also the pixels surrounding that pixel. For instance:

```
eval "B=A+A[-1,0]+A[0,-1]+A[1,0]+A[0,1]"
```

Adds the pixel and its four neighbors for each pixel in the image and stores the result in the correct location in image B. In this expression 'A[-1,0]' indicates the left-hand neighbor of the pixel and 'A[0,1]' is the pixel below. The first number is the x distance from the pixel and the second number is the y distance. When using offsets like this in an expression, problems occur at the borders of the image, because the pixels on the edge do not have a neighbor on one or two sides. For this situation, the second parameter of **eval** ('border value') is used. If 'border value' equals -1 then the edge will work like a mirror when a pixel outside the image is pointed to, a pixel within the image will be used. The following example should make this clear.:

```
eval "D=A[-256,0]" -1
```

It should be clear that the offset is mirrored in the border of the image. If however 'border value' is specified to be any other number (say 5) then no mirroring will take place, but all pixels addressed outside the image will have the specified value. To illustrate:

```
eval "C=A[-256,0]" 5
```

Aside from the use of operators listed in Table 6-3, **eval** can be used to evaluate a number of mathematical functions in the expression. For instance:

```
eval "C=sqrt(A)"
```

calculates the square root of every pixel in image A and stores the result in image C.

Functions can be mixed freely with operators:

```
eval "D=sqrt(A*255)"
```

The result of the operation 'sqrt' is a float so the resulting image will be a **FLOAT_2D** image. If, however, a **GREY_2D** is wanted, the data can be converted by using the function 'irint' in the expression:

```
eval "D=irint(sqrt(a*255))"
```


The following functions may be used with **eval** :

<i>function</i>	<i>description</i>	<i>type of result</i>
abs(x)	absolute value of x	float
acos(x)	arc cosine of x	float
asin(x)	arc sine of x	float
atan(x)	arc tangent of x	float
atan2(x,y)	arc tangent of x/y	float
cbrt(x)	cubic root of x	float
cos(x)	cosine of x	float
exp(x)	to the power of x	float
hypot(x,y)	square root of (x*x + y*y)	float
irint(x)	returns x as a integer	integer
log(x)	natural logarithm of x	float
log10(x)	base logarithm of x	float
log2(x)	base logarithm of x	float
max(x,y)	maximum of x and y	float
min(x,y)	minimum of x and y	float
pow(x,y)	x to the power of y	float
rnd()	random number	integer
sin(x)	sine of x	float
sqrt(x)	square root of x	float
tan(x)	tangent of x	float

Table 6-4: functions in eval()

In addition to the operators and the functions in **eval**, two keywords are available for special use. The words 'xx', 'yy' and 'zz' represent a pixel's position in an image, so to fill an image with a grey ramp:

```
eval "C=yy"
```

The pixels in image C now all have a value equal to their y position in the image.

The last operator in the table is the query operator (' ? : '). This operator is used for testing and taking different actions depending on the result of the test. To use the operator, three different parts need to be specified. The first part (before the question mark) must be the test, the second part (after the question mark and before the semi colon) is the part that is executed if the test is passed (true). The third part (after the semi-colon) is executed if the test failed. This construction should be well known to C programmers.:

```
readfile trui B
```

```
eval "C=xx<128?A:B"
```

Any of the three parts may be any expression accepted by **eval**, so statements of the following sort are acceptable:

```
eval "D=xx<128?A[128,0]:255-B"
```

Storing images on disk

The images used thus far were read from disk into memory using the command **readfile**. Most images will have to be read from disk before they can be processed. Because there are several ways to create an image, (cameras, microscopes etc.), the data format must be known in order to read it. The dimensions and image type must also be known. For this reason several 'standard' image file-formats have been developed. Currently, SCIL_Image supports the following file-formats for images:

ICS format

TIFF format

JPEG format

TCL format

AIM format

Because all SCIL_Image image types (2-D and 3-D) can be stored using the **ICS** format, we use this as our primary format. We recommend this format to store images produced using SCIL_Image.

The ICS format

With the ICS format image data is stored in two files. One file contains the actual data of the image and has the extension **'ids'**, The other file contains header information, such as image type and dimensions, and has the extension **'ics'**. This header file is an ASCII file which can be read and edited using standard tools. Aside from information about the type and dimensions of an image, data regarding the resolution of the image, date of creation and other pertinent details may be stored in the image header file. See the appendix "ICS file format" for a full description of this image file-format.

The TIFF format

TIFF stands for Tag based Image File Format and is designed for the interchange of digital image data. The format claims to be independent of any specific computer, operating system, filing system, compiler etc. Because of this independence the file format is widely spread. A large number of specifics of an image can be stored in a TIFF file, all meant to enable an exact reproduction of the image to be made in a different place than it was made. The exact description of the format is not the goal of this manual and thus omitted. The recommended extension for this format is **'tif'**, all extensions that start with these three characters are

accepted. SCIL_Image is able to read and write TIFF images that comply with the TIFF 6.0 specifications.

The JPEG format

The JPEG image file can be used for grey 2D and color 2D images in SCIL_Image. The JPEG standard is designed as a compression standard for continuous-tone still images. The image data is compressed “lossy” upon storage, meaning that no exact representation of the data can be made when the data is retrieved from file again. The advantage of “lossy” compression is the large compression ratio that can be achieved. The JPEG standard does include a method for “lossless” compression but this is not (yet) available. The JPEG format supported by SCIL_Image is actually the “JPEG File Interchange Format” (JPEG FIF also known as JFIF), which is the standard JPEG format with a few additional requirements. This may cause some JPEG files to be rejected by SCIL_Image but most JPEG files will be read. SCIL_Image requires that JPEG files have the `.jpg` extension.

The TCL format

The TCL(-Image) format is supported for historical reasons and can only be used for grey valued 2-D images. One file is used per image containing both the image data and the header information about the dimensions of the image. Comment can also be stored in this file. One disadvantage of this format is that not all image types under SCIL_Image can be stored in this file-format. Another is that the header information is not in ASCII and therefore cannot be read or modified easily. Although the format does not have an obligatory extension after the filename, SCIL_Image only accepts files that have the `.dat` extension, which is therefore the recommended extension, for this format. We insist on this extension because we also support other formats with other extensions.

The AIM format

The AIM format is also supported for historical reasons (we only support reading files in this format, it cannot be written). This format consists of two files per image, a header-file and a data-file. The header is not in ASCII, so it cannot be easily read or modified. The header-file of this format has the extension `.hd` and the datafile the extension `.im`. If the header file is not present the dimensions 256 * 256 are assumed.

Non image data (var_objects)

Var_objects in SCIL_Image are objects that are used to store non-image data. Several measurement functions in SCIL_Image store their result in a var_object. In fact, a var_object is equivalent to an array in C. It was introduced to prevent inexperienced users from having to declare and address arrays. They are also convenient for performing checks on sizes and data-types.

Behavior of var_objects

A var_object is similar to an image in the sense that the size and type of a var_object are automatically adjusted if an operation returns a specific type and size. A var_object can have a number of dimensions ranging from 0 to 5, so a variety of data can be stored in it, from a scalar (0-dimensional) to a multidimensional array (maximum 5 dimensions). Further, a dimension called the 'nr_channel' (number of channels) is added for compatibility with existing image types. This is useful for instance, for complex images, where each pixel has two values associated with it. Which var_objects are shown in the dialogue box can be controlled with a group name. Comments of arbitrary length can be attached to help the user in remembering what is stored in the var_object. Also the objects can be saved to disk.

By default four var_objects are available for the user, called obj1, obj2, obj3 and obj4. An unlimited number of extra var_objects may be created. Information about a var_object can be obtained with the function:

show_var_object_info obj1

This function results in a list of information about the object, appearing in the command window. For instance, the object's name, class, number of dimensions, sizes and pointer values will be displayed.

Datatypes of var_objects

The basic standard C types may be stored in var_objects. Further, the SCIL_Image type PIXEL (used in grey valued images) may be stored in a var_object. A complete list of all types is :

<i>type-name</i>	<i>description</i>
PIXEL	data type of GREY images (16 bits)
char	character (one byte)
short	short integer (16 bits)*
int	integer (32 bits)*
long	long integer (32 bits)*
float	floating point number (32 bits)
double	double precision floating point number (64 bits)

Table 6-5: var_object data-types

* integers in C may have a different number of bits, depending on the system and compiler used. The value used here is the number of bits used on most platforms.

Examples using var_objects

The var_objects are used in several different operations in SCIL_Image. Below, we show some examples

Example 1

A method to display a 3D image in 2D is to display the maximum pixel value of the pixels in corresponding locations in the various slices. In SCIL_Image, a function is available that can determine either the maximum value of the whole image or the maximum value along an axis. The result of this can be printed on the command window or stored in a var_object:

```
readfile chromo3d A
pix_maxval A obj1 Axis Z
show_var_object_info obj1
```

The information printed in the command window states that the type of data is PIXEL, the same type as the pixels of a grey valued image. Note that the var_object is 2 dimensional. This data can be put in to an image with:

```
var_object_to_image obj1 B GREY_2D
```

In image B the result of the operation, a good 2D representation of the 3D image is visible. The reason why the result of **pix_maxval** is not stored in an image directly is that the result can also be 1 dimensional or even 0 dimensional (a scalar), depending upon the dimensions of the input image, and the parameters of **pix_maxval**.

Example 2

In the above example the result of taking the maximum of a 3D image was a 2D array of pixels, made visible by putting the data in an image. The result of the same operation on a 2D image is a set of 1 dimensional data. This data can be used as input in other operations:

```
readfile orka256 A  
pix_maxval A obj2 Axis X  
lookup A B obj2
```

Example 3

The result of measurements in certain images can also be stored in a var_object:

```
readfile cermet A  
threshold A B  
invert_im B B  
label B C  
shape C obj1 0 0 obj2
```

The objects in image C have been measured. The number of each object is stored in var_object obj1 and the area of that object is stored in the corresponding element of obj2. These values can now be processed by other routines or be saved for later reference.

Storage of var_objects on disk

Like images var_objects can be stored on disk for later use. The format in which the data is stored resembles the ICS format. Again two files are used to store the object and its ASCII header. The data-file carries the extension '**vod**' (Var_Object Data) and the header-file the extension '**voh**' (Var_Object Header). The header is ASCII to enable easy reading and the comment attached to a var_object is also stored in that file.

Histogram objects

The histogram of an image (or a part of an image) can provide valuable information on the image. The information can be used for contrast enhancement, segmentation, statistical analysis and for normalization of the grey-tone deformations through the equalization command. SCIL_Image stores the histograms it calculates in so-called histogram-objects. The histogram-objects can be of any size and multi-dimensional (up to 5 dimensions). Stored in the object is not only the histogram-data itself but also the number of bins, the width of a

single bin and the median value of the lowest bin and the highest bin (all this for each dimension of the histogram).

The histogram can be made visible in separate window that can be resized and moved around by the **show_histogram** command:

```
readfile trui A  
show_histogram A
```

The histogram objects in SCIL_Image behave very much like images. When you choose a existing histogram-object to hold the histogram data of an image, the histogram object will automatically be reconfigured. For instance, if we calculate the histogram of a color-image, the histograms for the red, green and blue channel are shown separately:

```
readfile flamingo A  
show_histogram A
```

The histogram of any image-type can be shown (except of complex images) in any number of bins. Also, it is not necessary that all pixel values in the image are put in the histogram. Consider this example: a floating-point image containing values of between -1.0 and 1.0 of which only the data between 0.0 and 1.0 is shown in 100 bins:

```
convert A B float_2d  
sin_im B B  
show_histogram B 100 0.0 1.0
```

To inspect the contents of the histogram, a list of all values can be dumped to the terminal or stored in an ASCII file using the **dump_histogram** command:

```
dump_histogram histogram_of_B - 10
```

The command **show_histogram_info** gives some additional information about the histogram object, like number of bins, width of the bins and median value of the lowest and highest bin.

Chapter 7 Introduction to Image 2.1

This chapter introduces the Image library.

Read this chapter if:

- You want an overview on processing images with Image.
- You have basic knowledge of image processing. Knowledge of the C language is not required for this chapter.

What is Image2.1

Image 2.1 is a portable image processing software library written in the programming language ANSI C. It is currently available as a stand-alone C library on a variety of computer platforms (UNIX, PC MS-Windows, Macintosh).

Key features of Image2.1 are:

- a flexible image infrastructure,
- support of a large set of basic image types,
- an advanced and extensive set of image processing functions,
- fast implementations,
- abstract error and I/O handling,
- publish and subscribe mechanism.

The image2.1 library is not a complete image processing development environment as it lacks a "real" user interface and viewing capabilities. The library was developed to provide a framework for developing image processing software. The library does not contain system specific routines, and as such it is easily portable to other platforms.

Image infrastructure

The image library provides a framework to support a variety of image types. The infrastructure maintains a strict separation of image operations by type. The infrastructure provides a convenient and image type independent way to do image housekeeping. It deals with input/output adjustment of image operations and manages region of interest definition in images. The type mechanism simplifies the management of images and makes programming with different types of images easy.

Image types

Current supported image types are:

- Grey value 2D and 3D (16 bit/pixel),
- Binary 2D and 3D in packed format (32 pixels per word),
- Floating point images (single precision) 2D and 3D,
- Complex images 2D and 3D,

- Label images 2D and 3D to facilitate object measurements,
- Full color images 2D and 3D (24 bits/pixel).

For image input and output the image library supports TIFF, JPEG, ICS, TCL and AIM (input only) format.

Advanced and extensive set of image operations

The image library supports an extensive set of image operations. A large variety of filter and image manipulation routines are available. These include arithmetic operations, Fourier transforms, geometrical routines, a versatile image expression evaluator and image measurements.

Fast implementations

Special attention has been given to the implementation of fast algorithms and where possible filters have been decomposed, based on linear separable kernels. Fast and memory efficient mathematical morphology is available, comprising a complete implementation of mathematical morphology using arbitrarily sized and shaped structuring elements.

Abstract error and I/O handling

One of the design constraints of the image library was to maintain a strict independence of the user interface and platform. This guarantees the possibility to integrate the image processing software with every conceivable user interface and allows to run on a variety of systems. Therefore, there is no direct way to read from the input, or write to the output or error stream. The image library provides an abstract error stack mechanism to facilitate the error reporting. Also, a special function for text output is used throughout the image library. The textual output can be intercepted in any user interface by setting up a special output handler.

Publish and subscribe mechanism

To safeguard a complete separation between the application code and the (graphical) user-interface a publish and subscribe paradigm is used. Every object publishes important events to itself. Other objects interested in changes of that particular object may subscribe to it. As a publishing object does not explicitly know its subscribers, the separation between core application and user-interface is guaranteed. The advantage of publish and subscribe method is the possibility to extend the existing system with new user-interface behavior without having to change existing code. The image library has been implemented as a stand-alone imaging library with publishing image objects. This provides the developer with powerful

tools to construct a user-interface for the entire image library or allows to realize an end-user application without any modification to the image library.

The structure of this manual

In summary the remainder of this manual contains the following chapters:

- Chapter 7 gives a global introduction of the image library.
- Chapter 8 describes the publish and subscribe message passing mechanism and its use in the image library.
- Chapter 9 gives an in depth explanation of the image infrastructure and how to program with the image library.
- Chapter 10 describes the AIO library, analysis of images and objects. AIO supports facilities for particle measurement for shape and optical density such as in microscopic images of cells.
- Chapter 11 presents methods for fast morphological binary image processing using bitmapped representations of binary images.
- Chapter 12 explains how a new image type is created and integrated with the image infrastructure.
- Chapter 13 demonstrates how to create a stand-alone application using the image library by means of a sample application.

The need for Image2.1

The image library stems from the image processing environment SCIL-Image, which is a platform independent image processing system. SCIL-Image includes facilities which lack in the image2.1 library such as the user interface, viewing capabilities and facilities for interactive program building.

For many years it has been a valid approach to build applications within image processing environments such as SCIL-Image, and, run these applications within this system.

Nowadays, we witness the trend in application development shifting from self contained image processing environments towards integration with other applications. It becomes increasingly popular to include the image processing software as a component into more complex applications, or in other words as plug-ins into other applications. Furthermore,

creating end-user applications with its own user-interface and look-and-feel is essential for many companies and institutes.

Therefore, there is a clear need for a stand alone image processing library. The library provides a framework which makes it easy to develop image processing software. As the library contains no system specific calls and maintains a strict abstraction in the user interface, the library is platform independent and suitable to integrate into virtually every application.

Structure of the Image library

In the Image library, it was decided to maintain a strict separation of image processing functions by the type of image they work on. Associated with each image type are a number of type dependent operations, such as image arithmetic, and image I/O operations. The default supported image types are:

<i>image type</i>	<i>name in Image</i>
grey-valued 2-d	GREY_2D
grey-valued 3-d	GREY_3D
binary bitmapped 2-d	BINARY_2D
binary bitmapped 3-d	BINARY_3D
floating point 2-d	FLOAT_2D
floating point 3-d	FLOAT_3D
complex 2-d	COMPLEX_2D
complex 3-d	COMPLEX_3D
color 2-d	COLOR_2D
color 3-d	COLOR_3D
labeled 2-d	LABEL_2D
labeled 3-d	LABEL_3D

Table 7-1: image types available in Image

Writing your own image processing routines

Image provides a flexible object oriented image processing infrastructure. Image types are divided into class and sub-class of objects. Function overloading supports class specific operations. Through the Image infrastructure, a number of general purpose services are provided which support image type specific operation development. The type and dimensions of input images are checked. Output images are automatically converted to the correct type and dimensions based on the operation and input image(s). Furthermore, the infrastructure supports the processing of arbitrary shaped region of interest (ROI).

The chapter "Programming with Image" contains a detailed description of the infrastructure in the Image library. It also explains how to write new image processing functions which make use of the image data-structures of Image.

Custom Image types

Another strong point of the Image library is the ability of adding new image types without any changes to the infrastructure. By strict separation of the image types and the infrastructure not having any knowledge of the details of the image types, a new image type can be added without changes to infrastructure. The implementation of a new type is achieved by supplying a handful of pre-defined functions to the infrastructure.

The chapter "New image types" supplies in depth information about this feature. The topic is discussed by means of an example on how to implement byte-images (8bit) in Image.

Measurement of objects (AIO)

AIO is a framework for the Analysis of Images and Objects. It is designed to perform measurements of (microscopical) objects, gathering results in linked lists. These results can then be stored and used in databases (e.g. for statistical research). Objects are manipulated either based on the measurements or by pointing in an image (when using a GUI). The object data vector and the object image can be stored in a file and in an image silo respectively, for later retrieval. The combination of AIO and Image provides a flexible environment for (microscopical) image analysis, for both experimental use and application development.

In "**Analysis of Images and Objects (AIO)**", the capabilities of the AIO package under Image are described.

Binary images

In the implementation of the binary image types, the data structures used for image manipulation received careful attention, resulting in optimal performance for binary operations. By packing the binary data in a word, we are able to process 32 pixels in parallel. An important set of mathematical morphology functions, including erosions, dilations, and skeletons are implemented for arbitrary shaped structuring elements. The arbitrary shapes make the mathematical morphology functions useful for a broad class of image processing problems. To obtain optimal execution times, special implementations have been developed for the common isotropic erosion and dilation. Binary image processing functions are discussed in the chapter "**Bitmapped binary images**".

Appendixes

The appendixes of the manual contain additional information on Image.

Appendix: ICS file format description

The ICS format, used to store images on disk, is a proposed standard for the storage of image data. This format is described in detail in an article supplied in this appendix.

Chapter 8 Publish and Subscribe

This chapter is a description of the "publish and subscribe" message-passing mechanism and its use in the Image library.

General aspects

Publish and subscribe is a generic message-passing mechanism in which an object broadcasts messages about important events and changes regarding itself. Other (independent) objects can receive these messages as a means of getting information about changes to the publisher. The publishing objects do not know if and how many objects will actually get these messages and how they react upon them. Objects that want to receive messages from a publisher must subscribe to that particular object and messages will be sent to them only. This way handling events generates little overhead.

The Image library uses this mechanism to create a complete separation between the image-processing and the user-interface. Consequently, Image can be used with any user-interface (or none at all).

A Graphical User Interface (GUI) can use this "publish and subscribe" mechanism to create all kind of independent interface objects that react only to the object(s) to which they have subscribed. Adding new interface objects can then be done without the need to change the source code of the existing objects. It can even be done without having access to this code.

For example: A graphical user interface (GUI) will contain an image-viewer-object that is visualizing an image on the screen. When the image data changes and the image publishes this message, the viewer is notified and may act by visualizing the changed data. Just as easy a histogram-data-viewer can subscribe itself to the image and visualize the histogram of the image. These viewers do not know of each other's existence (and do not have to), just as the image does not need to know that two viewers are displaying its data. In a later stage without the image knowing it the viewers can "unsubscribe" themselves from the image and may attach themselves to another image.

Object requirements

For an object to publish events and allow other objects to subscribe to it, it (the structure) must contain a "void *publish;" field as its first member. The mechanism uses this field to store information regarding the subscribers to the object. An object-structure should look like this:

```
typedef struct my_object {
    void *publish; /* publish field */
    ...
    ... /* other fields */
    ...
}
```

The name for this field ("publish") is not mandatory, just recommended.

Subscribing and unsubscribing to objects

To receive the messages broadcasted by a publishing object the receiving object must subscribe to that object. For that it must use the function `spb_subscribe()`:

```
void spb_subscribe( void *object, void *ident, void
(*fun_ptr)(), void *client_data)
```

"object" is the publishing object.

"ident" is an identifier for the subscriber (typically a pointer to the subscriber).

"fun_ptr" is a pointer to the function the subscriber uses to receive the messages.

"client_data" is an optional pointer to data the subscriber wishes to receive whenever the publisher broadcasts an event.

For a complete description of the function "fun_ptr" see "Receiving messages" on page 8-3. After subscribing to an object, the subscriber will receive all messages send by the publisher and may or may not react on each of the messages. After a while the subscriber can decide that it is no longer interested in the object and wishes to unsubscribe. For that it calls the function `spb_unsubscribe()`:

```
void spb_unsubscribe( void *object, void *ident, void (*fun_ptr)())
```

"object" is the publishing object it is subscribed to.

"ident" is an identifier for the subscriber itself.

"fun_ptr" is a pointer to the function the subscriber used to receive the messages.

The subscriber then is disconnected from the object and it will no longer receive any messages from the object. To uniquely identify itself to the mechanism, it must provide exactly the same identifier and function pointer as when it subscribed itself to the object.

Receiving messages

The function the subscriber uses to receive the messages from a publisher must have the following header:

```
void func( void *object, void *ident, int mess, void *data, void
*client_data)
```

"object" is the publishing object.

"ident" is an identifier for the subscriber.

"mess" is the numeric ID of the message being published.

"data" is (a pointer to) the message specific data (if any).

"client_data" is (a pointer to) the data specified when subscribing to this object.

It is the responsibility of the subscriber to recognize the messages and act upon them correctly. The subscriber must silently ignore unrecognized and uninteresting messages as these may be of interest to other subscribers to this object.

Publishing messages

An object that wishes to publish messages (and meets the structure requirement as mentioned in "Object requirements" on page 8-2) calls the function `spb_publish()`:

```
spb_publish(void *object, int message, void *data)
```

"object" is the publishing object itself.

"message" is the numeric event ID

"data" is a pointer to the (optional) message specific data.

The type of the publisher (object) defines the messages published and the data accompanying those messages. This means that different types of data can accompany the same message when published by different object-types.

Processing messages

How to act when receiving a message is up to the subscriber. The "publish and subscribe" mechanism does not impose any obligatory actions to certain messages except for the "SPB_DESTROY" message as explained in "Messages" on page 8-5. However when processing messages, please keep the following in mind:

- When a publishing object has more than one subscriber, it is undetermined in which order these functions are called. The actual order may even change in future versions without notice.
- The current implementation of the publishing mechanism has a "depth-first" strategy. This means that new messages get priority. E.g.: Object B and object C are subscribed to object A and A publishes a message. Object B receives this messages and decides to publish an other message itself. This new message is first sent to the subscribers of object B before the message of object A is sent to object C.

Messages

In the include file "spublish.h" we give several often occurring events pre-defined message-IDs. Also a number of structures are defined for passing message data, we encourage you to use these whenever possible. Although we refer to most messages by name, you must remember that all messages are numbers that must be unique for that event-message.

One message deserves special attention, the `SPB_DESTROY` message. This message signals the destruction of an object. Because the information regarding subscribers is stored within the object itself (the `publish` field), the destroy message **must** be published before the object is actually destroyed. Another good reason for publishing the destroy message beforehand is the possibility that subscribers may still need to access the object.

Whenever a subscriber receives the `SPB_DESTROY` message, it knows that this is the final message from that publisher. The subscriber can not prevent the destruction of the publishing object. It must clean up everything that relates to that object and may even decide to destroy itself. It is not necessary to unsubscribe from an object that is destroyed, the mechanism will automatically unsubscribe all subscribers when an object publishes a `SPB_DESTROY` message.

Finally, the message values 0 to 9999 as well as the message prefixes `SPB_`, `IMP_` and `AF_` are reserved. If you wish to define additional messages for the use in an interface or elsewhere, use values outside the above mentioned range and symbolic names that do not use the above mentioned prefixes.

Publish and Subscribe in the Image library

The Image library uses the publish and subscribe mechanism for several types of objects. The images, the histogram objects and the error-object all publish events that happen. When you wish to subscribe to image and other objects, a problem arises. To subscribe to an image, you must have a pointer to that image. But when you write your source code, this pointer is not yet present. Image solves this problem by supplying an administrative top-level image that is permanently available. This top-level image publishes the creation of images.

Top-level publishes

As mentioned above, to obtain the pointers to images you wish to subscribe to, you must first subscribe to the top-level image. This top-level image is called "super_im" and is globally available as a void pointer or via the function `get_super_im()` :

```
void *s_im;  
s_im = get_super_im();  
spb_subscribe( s_im, NULL, funcname, NULL );
```

The pointer to this `super_im` can also be retrieved with the `get_super_im()` function. In the example above the function "funcname" receives a notification whenever new images are created. The function receives as its third parameter a message ID, in this case `SPB_ITEM_ADD` (see also "Receiving messages" on page 8-3). Its fourth parameter is a data pointer, which is the pointer to the new image. The message IDs are defined in the include file "spublish.h". Below a complete overview is given of all current messages of "super_im".

SPB_ITEM_ADD is published when a new image has been created, the data of this message is a pointer (`IMAGE *`) to the newly created image.

SPB_ITEM_DELETE is published when an image is about to be destroyed. The data is a pointer (`IMAGE *`) to the image. When this message is published, nothing can be done to stop the destruction of the image. The only reason this message is published beforehand is to prevent invalid memory access by subscribers that still reference the image because they do not yet know that the image no longer exists.

SPB_NEWTYPE is published when the type of image changes. The data is a pointer (`IMAGE *`) to the image that has changed. Regular images publish this message during the call to the `post_op()` function. However, Region Of Interest (ROI) images publish this message during the call to the `pre_op()` function.

SPB_RESIZED is published when the sizes of image changes. The data is a pointer (`IMAGE *`) to the image that has changed.

SPB_AUTO_DISPLAY is published when using the functions `don()`, `doff()`, `auto_display()`. The data is a pointer (`int *`) to a Boolean value that states if the auto display mode is on (1) or off (0). The auto display mode is considered to be a hint to the GUI to not display the images when they change. This feature is used to hide intermediate results of complex image processing operations.

Image publishes

After obtaining the pointer to an image, an object that wishes to monitor the changes of that image must subscribe to it using the `spb_subscribe()` function. The current messages that are published by image objects are:

SPB_DESTROY is published when an image is about to be destroyed. This publish is done right after the **SPB_ITEM_DELETE** message of the "super_im". It describes the same event of the same image. It also is published prior to the actual destruction of the image, which cannot be halted. The data with this message is a NULL pointer because the image is identified already by the publishing object (first parameter of the subscriber's function).

SPB_CHANGED is published when the image data changed. The data with this message is a pointer to a `spbAREA3D` structure. This structure describes the region that changed; the offset and the size. A NULL pointer is supplied when the entire image changed. This happens with most image processing operations.

SPB_NEWSTATE is published when the Image Flags changed. The data is a pointer to a `spbNSTATE` structure that contains both the old and the new value of the Image Flags.

SPB_NEWCLUT is published when the color lookup table (CLUT) of the image changed or was replaced by another. The data is a pointer (`CLUT *`) to the CLUT.

SPB_NEWTYPE is published when the type of the image changed. The data is always a NULL pointer. The new type can be retrieved from the image itself. This publish is followed by a **SPB_CHANGED** publish of this image because the contents changed as well.

SPB_MOVED is published by a ROI only, it signals that its position inside the parent changed. The data is a NULL pointer, the new location can be retrieved from the image itself.

SPB_RESIZED is published when the sizes of the image changed. The data is a NULL pointer, the new sizes of the image can be retrieved from the image itself. This publish is followed by a **SPB_CHANGED** publish of this image because the contents changed as well.

SPB_CREATE_DISPLAY is published as a request to the display-interface (if present) to create a display window for this image. The data is a `spbAREA3D` structure containing the desired location and sizes of the display-window.

Color-lookup table publishes

In Image 2.1 a color lookup table (CLUT) may be an integral part of an image. The definition of the CLUTs allows a display-interface to store some window-system dependent data in the CLUT. To give a display interface the opportunity to react to changes to the CLUTs, these

changes are published via a top-level CLUT. This top-level CLUT is called "super_clut" and is globally available as a void pointer or via the function `get_super_clut()`.

```
void *s_clut;
s_clut = get_super_clut();
spb_subscribe( s_clut, NULL, funcname, NULL );
```

It currently publishes the following messages:

SPB_ITEM_ADD is published when a new clut object is created. The data is pointer (CLUT *) to the new clut.

SPB_ITEM_DELETE is published when a new clut object is created. The data is pointer (CLUT *) to the clut. When this message is published, nothing can be done to stop the destruction of the clut. The only reason this message is published beforehand is to prevent invalid memory access by subscribers that still reference the clut because they do not yet know that the clut no longer exists

SPB_CHANGED is published when the contents of a clut is changed. The data is a pointer (CLUT *) to the clut. All functions that somehow change the contents of a clut must publish this message.

Top-level Histogram publishes

Like the images, the histogram objects publish events to allow for a flexible way of handling histograms. Similar to images, a "hook" is permanently present to provide a way to detect the creation of new histograms. This top-level histogram is called "super_histo" and is globally available as a void pointer or via the function `get_super_histo()`.

```
void *s_histo;
s_histo = get_super_histo();
spb_subscribe( s_histo, NULL, funcname, NULL );
```

The messages the top-level currently publishes are:

SPB_ITEM_ADD is published when a new histogram object is created. The data is pointer (HISTOGRAM *) to the new histogram.

SPB_ITEM_DELETE is published when a histogram object is about to be destroyed. The data is a pointer to the concerning histogram. This message is published just before the actual destruction of the object. The destruction of the object can not be stopped.

Histogram publishes

The histogram objects in Image publish the following messages:

SPB_DESTROY is published when the object is about to be destroyed. The data with this message is always a NULL pointer.

SPB_CHANGED is published when the histogram data changed. The data with this message is a NULL pointer.

Error stack publishes

In the Image library, an error-stack is present to log the location and messages of an error occurring somewhere in the low-level of the image processing functions. When an error occurs, the relevant functions handle it and all add their error values and messages to this stack. When all these functions have aborted, the error stack publishes a **SPB_CHANGED** message to signal to the interface that an error has taken place. The stack is a global structure called `im_error_stack`, a pointer to it can be retrieved using the function `get_im_error_stack()`. The type of the structure is `IM_ERROR_STATUS` that is defined in the include file `"im_error.h"`. Please note that this variable is a structure and in order to subscribe to it, you must use the address of this structure!

The correct subscribe call for the error-stack therefore is:

```
#include "spublish.h"
#include "im_error.h"

IM_ERROR_STATUS *estack;

estack = get_im_error_stack();
..
spb_subscribe( estack, .. /* other parameters */);
```

For further information regarding this error stack, please refer to the chapter "**Programming with Image**".

Chapter 9 Programming with Image

This chapter describes how to write image processing functions for use in Image

Do not read this chapter if:

- You are a novice user
- You want to use existing routines only
- You have no experience with programming in C

Read this chapter if:

- You are an experienced user
- You want to write your own image processing routines
- You want to know more about the image processing infrastructure

Introduction to Image

This chapter is on developing image processing routines and incorporating them into an image-processing application that uses the Image Library. A solid knowledge of the C language is assumed. This chapter contains the information needed to add image processing functions to the library or to modify existing operations. For a developer, an important feature of Image is the support of different image types. Knowledge of the infrastructure is therefore essential for creating functions that make maximal use of the library. An overview is presented in "Image infrastructure" on page 9-3.

A typical routine in the Image context reads as follows:

```
function heading(images [, parameters])
declaration of an image data structure /* "The IMAGE structure" */
calls to parameter checking routines /* "Checking routines" */
call to pre_op(.) /* "Dynamic adjustment (Pre_op, Post_op)" */
... processing ... /* "A simple example" */
... error handling ... /* "Error handling and reporting" */
call to post_op(.) /* "Dynamic adjustment (Pre_op, Post_op)" */
function ending
```

The **IMAGE** data structure is discussed in "The IMAGE structure" on page 9-6. In "Dynamic adjustment (Pre_op, Post_op)" on page 9-12 the (obligatory) `pre_op()` and `post_op()` routines are considered as well as the dynamic size and type adjustment facilities. An example of an image processing function is found in "A simple example" on page 9-19. Routines for checking non-image parameters are described in "Checking routines" on page 9-23. "Error handling and reporting" on page 9-21 is devoted to error handling.

For textual output, a special function is defined, intended to replace the default `printf()` and `fprintf()` functions, thus enabling a User-interface to influence the presentation of all text output. The section "Textual output" on page 9-26 discusses this subject.

The function overloading mechanism described in "Function overloading" on page 9-27 is a useful tool for keeping functions simple which must implemented for multiple image types. In "Data conversion (convert)" on page 9-32, the design of routines which involve a conversion of the image type is considered.

When dealing with numerical non-image data, the `var_objects`, described in "Var_objects" on page 9-37, can be useful. Using **HISTOGRAM** objects, which can hold multi-dimensional and arbitrary length histograms is discussed in "Histogram objects" on page 9-40.

Image infrastructure

The infrastructure is designed to be independent of specific image types in order to provide a consistent framework which can support a growing variety of image types. Image types can be divided into classes and subclasses. For this purpose two labels are associated with each image type. These labels can be found in the IMAGE-structure (see "The IMAGE structure" pg. 9-6). The first is called `type_ident` (type identifier) and has a unique value for each image type. The second is called `type_spec` (type specifier) and specifies the data representation and layout of the image type. The number of classes is limited to 32. The number of subclasses is unlimited. The image types, their type-idents and type-specs which are currently implemented are

<i>image type</i>	<i>type_ident</i>	<i>value</i>	<i>type_spec</i>	<i>value</i>
grey valued 2-d	GREY_2D	(1)	G_2D_SPEC	(1)
binary 2-d	BINARY_2D	(2)	B_2D_SPEC	(2)
floating point 2-d	FLOAT_2D	(3)	F_2D_SPEC	(4)
complex 2-d	COMPLEX_2D	(4)	C_2D_SPEC	(8)
grey valued 3-d	GREY_3D	(5)	G_3D_SPEC	(16)
binary 3-d	BINARY_3D	(6)	B_3D_SPEC	(32)
floating point 3-d	FLOAT_3D	(7)	F_3D_SPEC	(64)
complex 3-d	COMPLEX_3D	(8)	C_3D_SPEC	(128)
color 2-d	COLOR_2D	(9)	COLOR_2D_SPEC	(256)
color_3d	COLOR_3D	(10)	COLOR_3D_SPEC	(512)
labeled 2-d	LABEL_2D	(33)	L_2D_SPEC	(G_2D_SPEC)
labeled 3-d	LABEL_3D	(34)	L_3D_SPEC	(G_3D_SPEC)

Table 9-1: present image types.

Each `type_ident` has a unique value which is used to identify the image type. The label `type_spec`, can be the same for more than one image type, as is the case for labeled and grey valued images. They have the same `type_spec` because both the data representation and layout are the same for both image types.

Functions which operate on images of type `GREY_2D`, are also capable of processing also those of type `LABEL_2D`. `GREY_2D` is a class and `LABEL_2D` is a subclass of it. Due to the interpretation of the data however, not all operations that are meaningful on a `GREY_2D` image are valid on a `LABEL_2D` image and vice versa. For example, a median-filter is a useful operation to perform on a grey valued image for purposes of noise reduction, but performing it on a labeled image is of little value.

A table of available operations for each image type is maintained to keep track of which operations are admissible for that type. The overloading mechanism is used to apply the appropriate function based on the image type. Therefore, functions which perform type

specific operations can be kept relatively simple since they need only consider the specific characteristics of one image type. Also not having to change existing source code when implementing an operation for another (new) image type, has considerable advantages. This is explained in depth in "Function overloading".

The Image types

In Image of each basic image type a 2 dimensional and a 3 dimensional variant is present. For each of these image types the pixels are stored consecutively in memory. First the pixel of the first row, then the pixels of the second row etc. The 3D image types are laid out in memory as a set of 2D images, first all the pixels of the first Z-slice, then all the pixels of the second Z-slice, etc.

Grey valued images

Grey valued images consist of pixels, each of which contain an integer value. These pixels are represented in Image by a in the C language defined datatype 'PIXEL'. This type is defined to be a 'short int', which means that, on most computers, 16 bits (= 2 bytes) per pixel are used. Therefore the value ranges from -32768 to +32767 (a total of 65536 values).

Binary bitmapped images

Each pixel in a binary images has value 0 or 1. One bit is therefore sufficient for the storage of each pixel. In Image we store thirty-two binary pixels in each 32bit integer (long word) so that a binary image requires only a fraction of the space occupied by a grey valued image. A border of long words is reserved around each binary image to enable very fast morphological operations

Floating point images

Floating point images consist of pixels each of which can have any real value within precision limitations. Each pixel is represented by the C type 'float', which on most computers occupies 32 bits. Though floating point images can contain more information than integer valued images, they are far less common. This is because computers used to work far more efficiently with integers, and because integer valued images require half the space (16 bits per pixel) and most image acquisition happens in integer valued format.

Complex images

Each pixel in a complex image is a complex number and therefore has a real and imaginary part. In Image, each of the parts is represented with the C type 'float'. This type of image is specifically required for the Fast Fourier Transforms (FFT).

Labeled images

Like a grey valued image, a labeled image is made up of integers with the C defined datatype 'PIXEL'. The interpretation of the values in a labeled image is however entirely different. The information in a labeled image is similar to that in a binary image. The pixels in the image either belong to an object or to the background. In a labeled image, however, each object in the image has its own number or 'label', used to distinguish it from other objects in the image. The pixels in an image which have been determined to be part of an object, and are connected, are all assigned the same unique label. Each pixel in a labeled image has a value identifying it as a member of a set or object. Labeled images allow measurements such as size and shape to be performed on the image objects.

Color Images

In Image color images can be represented by five different color-models. The present models are RGB, (CIE) XYZ, (CIE) Lab, HSI and CMYK. The color models are each represented in a different manner:

RGB : each pixel is made up of a **Red**, **Green** and **Blue** component, each component (channel) occupies 1 byte in memory (C type : unsigned char). The three channels together with a fourth (extra and empty) byte are stored in one C structure.

CIE XYZ : each pixel consist of **three** floating point values, **X**, **Y** and **Z**, that are stored in one C structure. Each channel occupies 4 bytes of memory (C type : float).

CIE Lab : each pixel consist of **three** floating point values, **Lightness**, **a** and **b**, that are stored in one C structure. Each channel occupies 4 bytes of memory (C type : float).

HSI :each pixel consist of **three** floating point values, **Hue**, **Saturation** and **Intensity**, that are stored in one C structure. Each channel occupies 4 bytes of memory (C type : float).

CMYK : each pixel consist of **four** floating point values, **Cyan**, **Magenta**, **Yellow** and **K** (sometime also named **Blackness**), that are stored in one C structure. Each channel occupies 4 bytes of memory (C type : float).

The IMAGE structure

All images are referenced through a pointer to an IMAGE structure regardless of the type of the image. The actual data of an image in Image is stored in a contiguous piece of memory. An image pointer is type omnipotent in the sense that the referenced image may change type but the image pointer is retained. The IMAGE structure consists of two parts. The first part contains information that is for all image types. One of its members points to the second part which is image type dependent. The second part is called the *image type descriptor*.

The general part of the IMAGE structure :

```
typedef struct image_t {
void   *publish;           - Publishing hook
int     type_ident;        - Type Identifier
long    type_spec;        - Data structure specification
int     lenx;             - Image Width
int     leny;             - Image Height
int     lenz;             - Image Depth
int     slice;            - Slice that is displayed (3-d only)
int     o_lenx;           - Old Image Width
int     o_leny;           - Old Image Height
int     o_lenz;           - Old Image Depth
char    image_name[IM_NAMELEN]; - Image Name
char    window_name[IM_NAMELEN]; - Display Window Name
void    *clut;            - Image color lookup table
void    *graphics;        - Image Graphics Specific Info
ROI     *roi;             - Region of Interest Descriptor
void    *in_descript;     - Image Type Descriptor (input)
void    *out_descript;    - Image Type Descriptor (output)
long    flags;            - Image Flags
int     op_cnt;          - Operation Counter
void    *image_info;     - Image Info List
void    *future4;        - Pointer for future use (do not use)
} IMAGE;
```

The members of the IMAGE structure can be referenced through macros for convenience.

These macros also reside in the file 'image.h'. They are:

```
ImageTypeIdent(ip)      (ip)->type_ident
ImageTypeSpec(ip)      (ip)->type_spec
ImageWidth(ip)         (ip)->lenx
ImageHeight(ip)        (ip)->leny
ImageDepth(ip)         (ip)->lenz
ImageSlice(ip)         (ip)->slice
OldImageWidth(ip)      (ip)->o_lenx
OldImageHeight(ip)     (ip)->o_leny
OldImageDepth(ip)     (ip)->o_lenz
ImageName(ip)          (ip)->image_name
ImageDispName(ip)     (ip)->window_name
ImageClut(ip)          (ip)->clut
ImageGraphics(ip)     (ip)->graphics
ImageROI(ip)           (ip)->roi
ImageIn(ip)            (ip)->in_descript
ImageOut(ip)           (ip)->out_descript
ImageFlags(ip)         (ip)->flags
ImageOpCount(ip)      (ip)->op_cnt
ImageInfo(ip)          (ip)->image_info

ImageSize(ip)          ImageWidth(ip)*ImageHeight(ip)*ImageDepth(ip)
```


The first two fields of any image type descriptor have a fixed meaning and are therefore present in a separate structure that can be referred to:

```
typedef struct type_fixed {
    int    type;          - Type Identifier
    void   *data;        - Pointer to start of Image Data
} IMAGE_TYPE_FIXED;
```

The macros to access the fixed members of any image type descriptor:

```
ImageInType(ip)  ((IMAGE_TYPE_FIXED *) (ip)->in_descript)->type
ImageOutType(ip) ((IMAGE_TYPE_FIXED *) (ip)->out_descript)->type

ImageInData(ip)  ((IMAGE_TYPE_FIXED *) (ip)->in_descript)->data
ImageOutData(ip) ((IMAGE_TYPE_FIXED *) (ip)->out_descript)->data
```

The meaning of remaining fields of the type descriptor depends on the image type. An effort has been made to keep them as similar possible to simplify the programming task. An example of a type specific descriptor (also in the file 'grey_2d.h'):

```
typedef struct grey_2d_t {
    int    type          - Type Identifier (TYPE_FIXED)
    short *data;        - Pointer to Image Data (TYPE_FIXED)
    int    lenx;         - Image Width
    int    leny;         - Image Height
    int    plane;        - Plane specifier
} GREY_2D_IMAGE;
```

The other type descriptors can be found in the include files for the different image types ('grey_3d.h', 'float_2d' etc).

In the general part of the IMAGE structure the members 'in_descript' and 'out_descript' are pointers to image type specific descriptors. The reason for using two type descriptors is discussed in "Dynamic adjustment (Pre_op, Post_op)" on page 9-12.

Image Flags

The flags field of the IMAGE structure is used to store some 'special treatment' flags for the image. Each flag occupies one bit of the 32 bit long word. Currently these flags are defined:

- | | |
|----------------------|---|
| READ_ONLY | (bit 0, integer value 1), image is read-only, the infrastructure does not allow you to use the image as an output image. The image cannot be altered of size and/or type. |
| NOT_IN_DIALOG | (bit 1, integer value 2), signals to the GUI that the image should not be shown in dialog boxes. |
| NO_AUTO_POINT | (bit 2, integer value 4), signals to the GUI that when pointing in the image viewer with the mouse, the pixel information should not be shown. |

NO_AUTO_DISPLAY (bit 3, integer value 8), signals to the GUI that the image should not automatically be displayed on changes to the image.

All other bits in the field flags are currently not used but reserved for future use.

Region of interest data structure (rectangular)

When writing image processing routines in Image, the programmer does not have to deal with regions of interest (ROI). The infrastructure deals with ROIs in such a way that if a ROI is specified to be the input or output for an operation, it can be treated as a normal image by the routine. A ROI is in fact a small image that exists as a separate image only during operation. The infrastructure also keeps track of possible conflict between the ROI and its parent. In the event that the parent image changes in size and as a result of that the ROI is no longer located entirely inside the parent image, an error is generated at the moment the ROI is used in an operation. And when the parent is destroyed, the ROI also gets destroyed.

A ROI in Image is defined by an normal IMAGE structure to which a small structure is added. An image is a ROI if the field 'roi' in the IMAGE structure points to a ROI structure (as defined in the include file 'roi.h'):

```
typedef struct roi_t {
    IMAGE      *parent;
    BOOL_MASK  *bool_mask;
    int        startx;
    int        starty;
    int        startz;
} ROI;
```

The information describing a ROI is scattered over several structures; an IMAGE structure, a ROI structure and the structures which also are used with normal images. Looking at the ROI structure, it can be seen that little additional information is needed to describe a rectangular or cubic ROI. Just the image in which the ROI resides (*parent) and the top left corner of the ROI in the parent image (startx, starty, startz). All other information is in the IMAGE structure itself. The field BOOL_MASK is used for defining non rectangular ROIs and is discussed in "Region of interest data structure (arbitrary shaped)" on page 9-9.

When a ROI is specified as the input or output image for an operation, the region is copied to a separate image by the `pre_op()` function. After the operation, `post_op()` copies the output result back into the parent image (see "Dynamic adjustment (Pre_op, Post_op)" on page 9-12).

Region of interest data structure (arbitrary shaped)

In addition to rectangular ROIs, support is provided for the creation of arbitrary shaped ROIs. The `BOOL_MASK` field in the ROI structure is used to define the shape. The `BOOL_MASK` structure (see 'roi.h') is defined as:

```
typedef struct bool_mask_t {
    BYTE      *data;
    int       lenx;
    int       leny;
    int       lenz;
} BOOL_MASK;
```

Every time an arbitrary shaped ROI is used, the data in the ROI is copied to a separate image (like rectangular ROIs) using a mask to distinguish the pixels that belong to the ROI from the other pixels. This mask is a Boolean mask which is similar to a binary image in the sense that all elements have a value of 0 or 1. A mask can be made by converting a `BINARY_2D` or `BINARY_3D` image into a `BOOL_MASK` using the function `get_bool_mask()`. This function returns a pointer to a `BOOL_MASK` that can then be passed the function `roi_define()`.

`BOOL_MASK` is a separate type in Image for several reasons. First, binary images contain (a lot of) data not needed for the mask. Furthermore it is necessary that the mask is present and valid for as long as the ROI exists. Using a binary image will cause problems whenever this image is, unintentionally, used in another operation. The mask structure is unlikely to be used accidentally in other operations.

It should be noted that when an arbitrary shaped ROI is used, the ROI-image is still rectangular and the pixels in the ROI-image that were not under the mask, are set to 0. The operation performed on the ROI, does not know whether it is an arbitrary shaped ROI or not and thus will operate on all pixels in the (rectangular) ROI-image. Especially histogram operations and statistical analysis on these pixel values may unknowingly be influenced by the 0-pixels.

Operation Counter

The `op_cnt` field of the `IMAGE` structure is a counter that is incremented each time the image is used as an output image for an image-processing operation. At several places in the image-infrastructure, this operation counter is used to test the validity of image related data. For instance the image-data of a ROI is copied only to the ROI-image if the `op_cnt` of the parent image indicates that the data of the parent image has changed. The `post_op()` function is responsible for incrementing the counters, so image processing functions that do not use `pre_op()/post_op()` on their output images can result in inconsistent ROI-data.

Image Info

The `image_info` pointer in the `IMAGE` structure can be used to store arbitrary, additional, image-related data with an image. By storing the data with the image, it is not needed to keep track of which data belongs to which image. A linked list of `IMAGE_INFO` structures is attached to the pointer. Each structure in the list points to user/application specific data. The data stored with each element of the list is neither interpreted nor processed by Image in any way. The user/application remains responsible for the contents of the data as well as the allocation of the memory (if needed). The infrastructure can destroy the data when the image is destroyed when required. The functions `AddImageInfo()`, `GetImageInfo()` and `RemoveImageInfo()` must be used to add the data-pointer to and retrieve and remove the data-pointer from the image. The `IMAGE_INFO` structure :

```
typedef struct image_info {
    char          infoname[IM_NAMELEN];
    unsigned long info_size;
    void          *dfunc)(void *);
    void          *infoptr;
} IMAGE_INFO;
```

`infoname` a string used to identify the data attached to this structure. The maximum length of the string is `IM_NAMELEN` (see 'image.h') characters (trailing '\0' included). Image reserves all names that start with **SI_** for private usage.

`info_size` size of the data attached, not yet used.

`dfunc` pointer to a (user/application specific) function to destroy the data when the image is destroyed. This function is called with `info_ptr` as the parameter. When the data cannot be destroyed (global variable) or should not be destroyed (shared with other images), a `NULL` pointer should be specified.

`info_ptr` pointer to the data. The data pointer to is not referenced or interpreted by Image in any way. Any type of data can be stored.

Image uses this mechanism to store the actual range of image data. Whenever the range of the image data needs to be calculated, the `get_pixel_range()` function is called, it checks if the range data is still valid by comparing the recorded operation counter with the actual operation counter of the image. When they are equal it is assumed that the minimum and maximum value stored in the image-info are still valid for the image, and consequently these values are returned instead of recalculating the image range again.

Image Color Lookup Tables

The `clut` field in the `IMAGE` structure can be used to attach a color lookup table (CLUT) to an image. Although most commonly used for influencing only the display of an image, some images (palette images) cannot be shown or interpreted without one. For now in Image the color lookup tables in Image are fixed length, RGB color model only:

```
typedef struct clut_t {
    char          clut_name[IM_NAMELEN];
    unsigned char r[256];
    unsigned char g[256];
    unsigned char b[256];
    unsigned long table[256];
} CLUT;
```

`clut_name` a string used to identify this lookup table. The maximum length of the string is `IM_NAMELEN` (see "image.h") characters (trailing `'\0'` included).

`r`, `g`, `b` arrays containing the RGB triplets specifying the colors.

`table` array that is reserved for use by the display interface attached. The interface can use this array to map the RGB triplets of the `r`, `g` and `b` array to the windowing system's colormap. The Image library does neither fill nor interpret the contents of this array in any way.

After initialization, the Image library supplies a number of "default" color lookup tables that can be attached to the images:

<i>name</i>	<i>contents</i>
<code>EMPTY_LUT</code>	all entries black
<code>BLUE_LUT</code>	a blue scale
<code>GREEN_LUT</code>	a green scale
<code>CYAN_LUT</code>	green and blue scale mixed
<code>RED_LUT</code>	a red scale
<code>MAGENTA_LUT</code>	red and blue scale mixed
<code>YELLOW_LUT</code>	red and green scale mixed
<code>GREY_LUT</code>	normal grey scale
<code>LABEL_LUT</code>	scale that simulates label display
<code>MULTI_LUT</code>	primary colors each in a specified bit
<code>OVERLAY_LUT_1...8</code>	specified bitplane red
<code>FALSE_COLOR_LUT</code>	false color table

Table 9-2 : pre-defined color lookup tables.

Custom tables can be made by using the `create_clut()` function, filling the `r`, `g` and `b` table with desired values. After that, the display interface must be told that the contents of the `clut` changed. This is done by publishing a **SPB_CHANGED** message to the super `clut` object:

```
void *sclut;  
CLUT *clut;  
  
sclut = get_super_clut();  
..  
/* creating an filling the clut.. */  
..  
spb_publish( super_clut, SPB_CHANGED, clut);
```

Dynamic adjustment (Pre_op, Post_op)

Frequently, in the course of image processing, routines are called which produce an output image of a different type or size than that of the destination image. In Image some tools are provided to perform the appropriate type and size conversions. The same tools handle ROI processing. By using them, the programmer is freed from converting images directly or from having to be concerned with whether an input (or output) image is actually a ROI.

The functions provided for these purposes are the `pre_op()` (**pre-operation**) and `post_op()` (**post-operation**) functions. Each image processing operation created for Image must call `pre_op()` prior to performing any processing on the images and `post_op()` when it is finished.

As an example, consider the threshold function which produces a binary image, and assume that the destination image is the input image. The `pre_op()` function will handle the creation of an intermediate data structure for output. If the image is simply converted to binary, then the grey value data will be lost. The `post_op()` routine displays the resulting image, and cleans up the intermediate data structures.

Though the conversion steps are admittedly simpler when operations are not performed in place, the programmer is freed from being at all concerned with image conversions and image display if the `pre_op()` and `post_op()` utilities are used. Further, keeping in style with other functions in Image, image processing functions should be designed so they can be performed in place, as is the case with all functions supported in the library.

In addition to handling image type and size conversions, these utilities also handle regions of interest correctly. If a ROI is handed to an image processing function rather than an image, the `pre_op()` function will make it possible for the function to treat it as if it were an image. The `post_op()` function will then clean up the necessary intermediate data structures.

In order to handle the conversions, the image type descriptors in the IMAGE data structure are used. Consider again the example of thresholding an image in place. The `pre_op()`

routine creates a binary image and connects it to the output image descriptor (`out_descript`). When the `post_op()` function is called, it destroys the data pointed to by the input image descriptor (`in_descript`) and the input image descriptor itself. It then connects the output image descriptor to the `in_descript` field of the IMAGE structure and displays the new image data.

The situation pointed out above looks like this. Before the call to `pre_op()`, the IMAGE structure has one descriptor attached to both `in_descript` and `out_descript`:

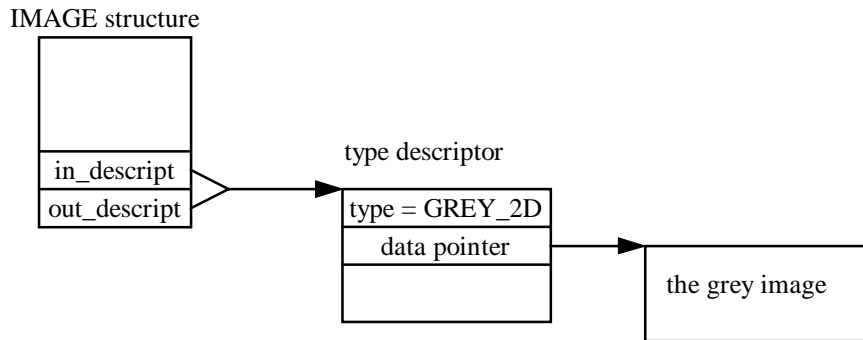


Figure 9-1: IMAGE structure layout before `pre_op()`.

Since the image had to change to type binary and yet remain grey value for input, `pre_op()` split up the `in_descript` and `out_descript` and attached different descriptors to them. The image now holds both the grey original and the binary destination:

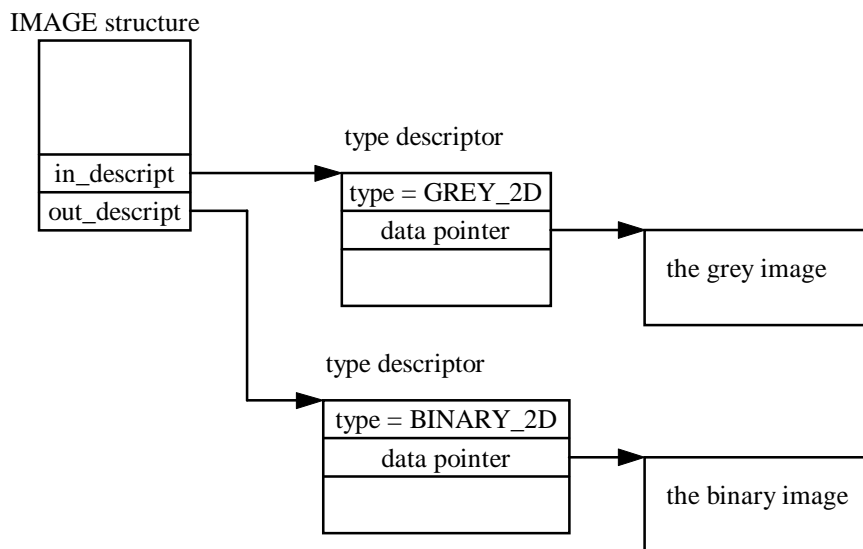


Figure 9-2 : IMAGE structure layout after `pre_op()`.

After the actual processing has been done, the grey image is no longer valid and is therefore removed by `post_op()`, resulting in a conversion of the image type. `post_op()` frees the memory of the grey image, throws away the input descriptor and attaches `in_descript` to `out_descript`:

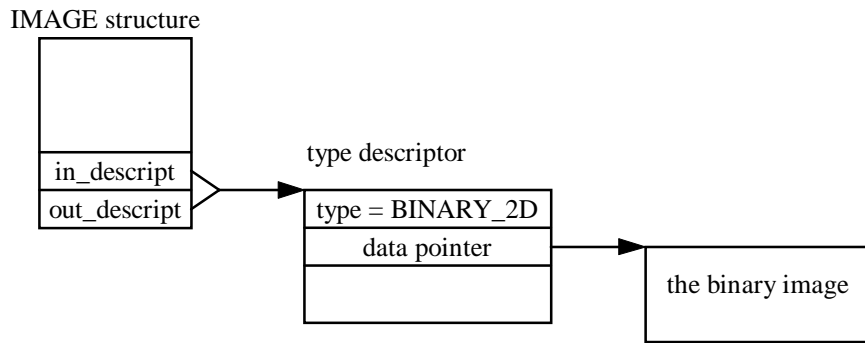


Figure 9-3 : IMAGE structure layout after post_op().

The fixed part of any type descriptor is necessary since it is possible for an IMAGE structure to have type descriptors of different image types simultaneously. Because each descriptor contains its type, the `post_op()` routine knows which descriptor needs to be destroyed after the operation.

The fixed data pointer allows the user to use the same macro to find the start of the image data regardless of image type.

The pre_op function

The `pre_op()` function is called with the following arguments:

```
int pre_op( IMAGE *first, IMAGE *second, int mode, unsigned long first_spec,
int second_ident )
```

The first two parameters are pointers to the images to be manipulated by `pre_op()`.

The third parameter tells `pre_op()` what to do. The mode can be either `COMPARE` or `ADJUST (ADJUST_NIP)` which are defines from 'image.h'.

The `first_spec` argument tells `pre_op()` the required image class of the first image. More than one class can be specified by applying the OR-operation (`|`) on the `type_specs` of the classes (see Table 9-1: present image types).

). This is useful when the same function can be applied to more than one image type. When the type of the image `first` is not important, the define `WHATEVER` can be used (all bits set).

The last parameter `second_ident` specifies the intended `type_ident` of the image `second`. This image then will be converted to that type if `ADJUST (ADJUST_NIP)` is specified and must be of that type if `COMPARE` is specified. This parameter may be specified as `OUT_AS_IN`, the meaning of which depends on the mode as explained below.

COMPARE mode

When mode is specified as COMPARE, `pre_op()` determines whether two images are of the same class and/or dimensions. It can also be used to check if one image is of a specific type. For instance when adding two images it is essential that they are of the same type and sizes. The call to `pre_op()` to check this would be:

```
pre_op(in1, in2, COMPARE, G_2D_SPEC|G_3D_SPEC,OUT_AS_IN);
```

`in1` and `in2` are the two input images. The valid classes are both `G_2D_SPEC` and `G_3D_SPEC` because the routine **add_im** can handle both 2d and 3d images. The last parameter is `OUT_AS_IN` because the second image `in2` has to be of the same class as the input image `in1`. So in this case the allowed types of images are `GREY_2D`, `LABEL_2D` (same class, see Table 9-1: present image types.

), `GREY_3D` and `LABEL_3D`. The function `pre_op()` itself returns OK (1) if the images are of the same class and have the same sizes. The fault status of `pre_op()` is NOT_OK (0). Because `pre_op()` handles ROI processing, all operations should call `pre_op()` because the specified image can always be a ROI.

For example, the routine that writes an image to disk (**writefile**) has only one image as input and can handle all types. The call to `pre_op()` therefore is:

```
pre_op( image, image, COMPARE, WHATEVER, OUT_AS_IN);
```

but could also be :

```
pre_op( image, image, COMPARE, ImageTypeSpec(image),  
ImageTypeIdent(image) );
```

Both calls have the effect that all image types are accepted by `pre_op()`. If the image is actually a ROI, it will be written to disk as if it were a normal image.

In the case that an operation has two input images which may be of different types but must be of the same sizes, the call to `pre_op()` would be:

```
pre_op( in1, in2, COMPARE, ImageTypeSpec(in1),  
ImageTypeIdent(in2) );
```

ADJUST(_NIP) mode

The `ADJUST(_NIP)` mode of `pre_op()` is more complex than the COMPARE mode because its behavior can be influenced by some global variables.

If `pre_op()` is called using `ADJUST`, the output image will be split up if the output image has to change type or size as is explained above. When the output image is of the correct type and size already, no new memory will be allocated and the input image descriptor and the

output image descriptor remain the same. When an operation is performed in place (input image is the same as the output image), the pixels will be replaced one by one as soon as each of them is processed. No reference to an original pixel value can be made once it is processed. For most operations, this is not a problem. Neighborhood operations, however, cannot be performed in place. To allow the same image to be specified as both input and output for a neighborhood operation, `pre_op()` can be used with `ADJUST_NIP` (`ADJUST Not In Place`). If `ADJUST_NIP` is used, the output image will be split up if the input image and output image are the same, even if no change of type or size is required. When this option is used, the output image will always be a different piece of memory, and the neighborhood operation can be performed with the same image as both input and output if necessary.

Output equal to input

The most common call to `pre_op()` is for adjusting the output image to match the type and sizes of the input image. For instance, in order to copy an image to another image, the output must be of the same type and sizes. The call to `pre_op()` to achieve this would be:

```
pre_op(in, out, ADJUST, G_2D_SPEC|G_3D_SPEC, OUT_AS_IN);
```

In this call, first the image `in` is checked if it is either a `GREY_2D` or a `GREY_3D` image. Then the image `out` is changed into the type and sizes of image `in`. If nothing went wrong, `pre_op()` will return `OK (=1)` and processing of the data can commence.

Output of specific type

If the output must be a specific type, the last parameter is used to specify the type. To threshold a `GREY_2D` image, the output image must be converted to `BINARY_2D`. To accomplish this:

```
pre_op( in, out, ADJUST, G_2D_SPEC, BINARY_2D);
```

Although the low-level routine (in this case) can handle both 2d and 3d, this call to `pre_op()` can only allow `GREY_2D` image for input. This is because the last parameter of `pre_op()` can only hold one image type at a time. The calls to `pre_op()` for 2d and 3d images must therefore be separate calls when the output image must be of a specific type.

Only output

Some operations only have an output image. It may be that the output image is required to be of a specific type. The `pre_op()` routine should then accept all types of image as input and convert the output image to the desired type. Assume the output image should be of type

GREY_2D, then either of the following calls to `pre_op()` will achieve the desired conversion.

```
pre_op( out, out, ADJUST, WHATEVER, GREY_2D);
pre_op( out, out, ADJUST, ImageTypeSpec(out), GREY_2D);
```

WHATEVER stands for all possible type specs and `ImageTypeSpec(out)` is the spec of image `out` and is therefore always correct. The `pre_op()` calls will therefore accept every type of image.

Type of input, sizes of output

In the above examples using `pre_op()` in ADJUST mode, the output image inherits the sizes of the image that is specified as the first argument. This is, however, not always desired, in which case special measures have to be taken. For instance, if the output image must be of the same type as the input image but keep its own sizes, the next two calls can be issued to achieve this behavior:

```
pre_op( in, in, COMPARE, G_2D_SPEC, OUT_AS_IN);
pre_op( out, out, ADJUST, ImageTypeSpec(out), ImageTypeIdent(in));
```

The first call checks if the input image `in` is of the correct type, in this case GREY_2D. The second call to `pre_op()` converts the output image `out` to the type of the input image `in` by specifying `ImageTypeIdent(in)` as the type to which the image `out` must be converted. Because the image `out` is specified as the first parameter, the sizes of `out` will be used. This results in the image `out` being converted to the type of image `in` with no change in size.

Special sizes

When the required sizes of the output image differ from those of both the input and output images, a special method is available to change the sizes. The function `set_cross_dim()` can be used to tell `pre_op()` that the next ADJUST action must use the supplied "special" sizes. The following call to `pre_op()` will set an image to a specific size:

```
#include "image.h"
#include "im_infra.h"

set_cross_dim( 100, 50, 4);
pre_op( in, out, ADJUST, G_3D_SPEC, FLOAT_3D );
```

The output image in this case will be a floating point image with the dimensions `x=100, y=50` and `z=4`.

ROI and pre_op

In addition to image size and type adjustment, the `pre_op()` and `post_op()` routines together also handle region of interest processing.

When a ROI is defined, a new IMAGE structure is reserved for it and processed as if it were an image. The only difference is that a ROI, does not have a display of its own. When a ROI is specified as input, `pre_op()` will copy the data from the parent (in the region of interest) to the ROI image. This way, the region of interest can be processed by low-level routines in the same fashion as other images. Note that with the introduction of the operation counter for images (see "Operation Counter" on page 9-9), the data of the ROI is only copied when the operation counter of the ROI-image is lower than the one of the parent image (the operation counter of a ROI can never exceed the counter of its parent). This prevents unnecessary copying of the data.

Using the ROI as an output image is only possible if it does not have to change size, because the ROI is nothing more than a part of another image. Its size is one of its primary attributes. If a call to `pre_op()` would result in a size change for a ROI, a warning will be issued, and the operation will be aborted. A call to `pre_op()` resulting in a type change for a ROI is no problem.

If a ROI is specified as both the input and output of an operation, `pre_op()` will copy the data from the parent to the ROI image as described above and change the type of the ROI if needed.

Multiple calls to pre_op

As explained above, after a call to `pre_op()`, an IMAGE structure can contain two different images. This is the case until a call to `post_op()` is made with the image. Before that time, other calls to `pre_op()` using ADJUST in which the same image is specified as being an output image, will fail. This can happen if in the routine, a call to another image processing routine is made after the `pre_op()` call, because, the other routine is also likely to call `pre_op()`. This must be prevented. Calls to other image processing routines must be made BEFORE the `pre_op()` function or AFTER the `post_op()` function.

The `post_op` function

A call to `post_op()` is issued at the end of each operation for each image that was specified as an output image in a call to `pre_op()` using `ADJUST(_NIP)`

`post_op()` requires one parameter:

```
int post_op(IMAGE *image)
```

`post_op()` cleans up after the actual calculations of a routine have been done. The function checks if the input and output descriptor of the specified image are different. If they are, it throws away the image that is connected to the input descriptor. `in_descriptor` will then be set to the same descriptor as `out_descriptor`, which is the result of the calculation. Finally it will publish that the image is changed, which is a signal for a Graphical User Interface to redisplay the image.

ROI and `post_op`

If the output of an operation is a ROI then `pre_op()` has already checked whether the result would fit in the ROI. `post_op()` then copies back the result to the parent image of the ROI and publishes that both the ROI and the parent image changed. If the ROI is of a different type than the parent, `post_op()` converts the parent to the same type as the ROI, and thus destroys the previous contents of the parent. This is done because the parent was implicitly specified as being the output image (by specifying the ROI as output).

A simple example

This section contains an example operation which calculates the maximum value of pixels at corresponding locations in two images and stores it in another image. The output image in the example is, in this case, of the same type as the input images.

Please note that the example is restricted to implementing the maximum function for the image types of `GREY_2D` and `GREY_3D` only. If a routine should operate on more image types, use of the function overloading facilities (page 9-27) is strongly recommended. The maximum operation:

```
#include "image.h" /* #1 */

int maximum( IMAGE *in1, IMAGE *in2, IMAGE *out) /* #2 */
{
    PIXEL *src1, *src2, *dest, maxi; /* #3 */
    unsigned long npix;

    if (!pre_op(in1, in2, COMPARE,G_2D_SPEC|G_3D_SPEC, OUT_AS_IN) ||
        !pre_op( in1, out, ADJUST, G_2D_SPEC|G_3D_SPEC, OUT_AS_IN))
        /* #4 */
        return(NOT_OK);

    src1 = ImageInData(in1); /* address of first image */ /* #5 */
    src2 = ImageInData(in2); /* address of second image */
    dest = ImageOutData(out); /* address of output image */
    npix = ImageSize(in1); /* number of pixels */ /* #6 */
    while (--npix) { /* #7 */
        if ( *src1 > *src2 ) /* which value is higher */
            maxi = *src1;
        else
            maxi = *src2;
        *dest = maxi; /* store biggest */ /* #8 */
        src1++; /* go to next pixel */ /* #9 */
        src2++;
        dest++;
    }
    return( post_op( out ) ); /* #10 */
}
```

Explanation of the function code:

- #1) The necessary data structures and definitions are in the include file 'image.h'.
- #2) The function takes three parameters which specify images, two of which are input images and one of which is an output image.
- #3) To access the data in the images, pointers are used. A counter (`npix`) is declared to control for the number of pixels processed.
- #4) Since three images are involved in the operation, two calls to `pre_op()` have to be made. The first one is to see if the two input images are of the same type and size. The second one is to adjust the type and size of the output image.
- #5) The address of the first pixel of each of the images is acquired through a macro (see "The IMAGE structure"). The addresses for the data in the two input images are accessed by the macro `ImageInData` and the address of the data in the output image is accessed by the macro `ImageOutData`. The reason why the output is accessed by a different macro is described in "Dynamic adjustment (Pre_op, Post_op)".
- #6) The number of pixels in an image can also be calculated using a macro. `ImageSize` multiplies the x, y and z dimensions of the image and returns the result.

- #7) The maximum is calculated for every pixel in the images. A `while` loop is used which terminates when all pixels in the image are processed. Within the loop, the corresponding pixels of the two input images are compared and the one with the highest value is stored in a variable.
- #8) The maximum value of the two pixels is stored at the corresponding location in the output image.
- #9) After a pixel has been processed, each pointer to the data is increased by one to point to the next pixel in the image. This continues until `npix` pixels have been processed.
- #10) After processing the data any temporary changes made by `pre_op()` should be cleaned up and the it must be published that the image-data has changed (see a interface can display the image). This is done by the call to `post_op()` with the image `out` as the argument. This function must be called for any image which has been adjusted using `pre_op()`. The value returned by `post_op()` is returned, so that the program that called the `maximum()` function can determine whether it completed successfully.

Error handling and reporting

An error can occur in any operation at any time due to several reasons. Once an error has occurred, the function must try to correct the problem. If it is unable to do so, it must return gracefully and signal its caller that an error occurred. It also must supply information about the nature of the error. To determine the location of an error and its nature accurately, Image supplies a mechanism to facilitate the error reporting.

Location of the error

To trace the location of an error, each function in the Image library participates in a stack mechanism that keeps track of the function that is being executed. For this purpose three functions are available:

```
void im_begin_func(const char *name )
void im_end_func(const char *name )
int im_report_error(const char *name, int val, const char *info)
```

The first statement of a function is a call to `im_begin_func()` with its name as the parameter. Just before the function exits in a no-error condition, it calls `im_end_func()` also with its name as the parameter. When no error occurs, the name-stack grows and shrinks as functions are being executed and returned.

If a function detects an error situation, this is reported to `im_report_error()` and an error-status is returned to its caller. It specifies its name, an error value and an optional information string as the parameters. Once an error is reported, the name-stack is copied to the error-stack and at that moment the exact location of the error is known. While all the functions in the stack handle the error and clean up, they store additional information in the error-stack. Other functions than the ones in the error-stack-trace are not allowed to store information in the error-stack, so only information regarding one error is stored. When all functions have returned, the error-stack publishes that an error has occurred and the user-interface can inform the user about the error if necessary.

Return values of functions

Functions written for use in Image should return a value based on their success or failure. The value can be examined when the function returns and its caller can take various actions depending upon the return value. Usually, if the function returns an OK status, the processing should simply continue. If a problem occurred, the status should indicate which problem arose. The program can either correct the error and continue or issue a warning and discontinue processing.

The value indicating success in Image is `IE_OK` which is defined as 1. Since various things may go wrong, a failure status may be indicated by one of many values, so that the programmer is able to indicate the cause of the error. In Image the failure status `IE_NOT_OK` (value '0') simply indicates that an error occurred but gives no information as to the cause. A negative value is used to indicate the cause of the error. Those used in Image can be found in the include file 'im_error.h'.

In some cases the negative error status can be ambiguous, if for instance a function returns some kind of measurement value, this value may well be negative even when no error has occurred. Other functions may be defined as being 'unsigned' (always positive), or returning a pointer. In these cases the caller can use the function `im_get_status()` to inquire if the called function exited with an error or with an OK status. `im_get_status()` will always return the error-value of the function that was most recently called by the current function provided that the function participates in the error-mechanism. When no error has been reported, `IE_OK` is returned.

Error handling

When detecting an error it is common practice to abort the normal processing of an operation and report the error to the user. Because of the independence of any user-interface, functions in Image have no means to visualize error-information. By using the error-mechanism

functions in Image supply the user-interface with as much information about the error as possible.

When an error is detected it is reported to the `im_report_error()` function. This initializes the error-stack (`IM_ERROR_STATUS` struct from `im_error.h`) by copying the name-stack to it and setting the error value for each function in the stack to `IE_NOT_OK (0)`.

The function that reports the error also supplies an error-value and a message, which are both stored in the stack. The function then returns with an error-status and its caller further handles the error-situation. In its turn it will clean up and add its own error-status and message to the error-stack with the `im_report_error()` function and also returns. In this manner all the functions in the error-stack handle the error and return.

Because the error-mechanism knows the exact location of the error, only the relevant functions can add their error-values and messages to the stack. If during the clean-up phase of a function, another error occurs and a function that is not in the error-stack reports this, the information regarding it will be ignored by the error-mechanism.

When the top-level function of the error-stack is reached, the error mechanism will publish that an error occurred. For an application this means that two ways are available to detect the occurrence of an error.

1. Checking the return value of the function it called from the Image library. This will be most common because the program can react to the error situation immediately when it occurs and that may be necessary to control the correct execution of the program.
2. Subscribing to the error-stack is typically done by the user-interface because it enables a uniform mechanism from a central location to visualize the occurrence of an error to the user.

Checking routines

Image provides a number of routines to check the values of non-IMAGE parameters. Each of these functions checks whether a particular requirement is satisfied and, if not, reports this as an error to the error-stack. If the requirement is met, `IE_OK (=1)` is returned, otherwise `IE_NOT_OK (=0)` is returned. These functions can be used in an `if`-statement to allow the programmer to take appropriate action, or prevent further processing if the routine fails.

The number of check routines is too large to discuss each of them here. An example of how they are used is shown below followed by a list of the routines provided. For a full description of these functions, the reader is referred to the (on-line) reference manual.

As an example, we consider a function `my_filter()` with the following header :

```
int my_filter( IMAGE *in, IMAGE *out, int size, int offset)
```

The function operates on images of type GREY_2D, size should be a value between 3 and 12 and offset may be any positive value. The function which check its input is as follows :

```
int my_filter( IMAGE *in, IMAGE *out, int size, int offset)
{
    im_begin_func("my_filter");
    if (!range_ok( size, 3, 12, "Filter size" ) ||
        !positive_ok(offset, "Offset" ) )
        return( im_report_error("my_filter", IE_NOT_OK,
                                "error in parameters"));

    if (!pre_op( in, out, ADJUST, G_2D_SPEC, OUT_AS_IN) )
        return( im_report_error("my_filter", IE_NOT_OK, NULL));

    /* rest of function */

    post_op( out );
    im_end_func("my_filter");
    return(IE_OK);
}
```

It is a sound practice to first check the validity of all the non-IMAGE parameters and when these are all O.K. then call `pre_op()`. This prevents the need to undo the `pre_op()` action if one of the other tests fails. Please note that the error mechanism checks the integrity of all images when an error has occurred.

In the example the first check is performed with `range_ok()` which checks whether the size parameter is between 3 and 12. If it is not in the specified range e.g. when size = 21, the following text is logged in the error-stack.:

Filter size [21] out of range (3..12).

The default error message in `range_ok()` is prefixed with the supplied text string ("Filter size"). If the check `range_ok()` succeeds, the check `positive_ok()` is issued. If this check fails this text will be logged:

Offset [<value of offset>] must be positive.

`pre_op()` checks the images and adjusts the output image to match the input image (see "The `pre_op` function" on page 9-14). If, for whatever reason, `pre_op()` detects that something is wrong, it will return with the value 0 and the operation will be stopped by the `return(im_report_error(...))` statement. If all checks are passed successfully, the operation will perform its calculations, followed by a call to `post_op()` for the image out.

Check functions

The available check functions are:

<code>image_ok</code>	checks if the pointer supplied is a valid image pointer
-----------------------	---

<code>images_ok</code>	checks if the two pointers supplied are valid images
<code>image_readwrite_ok</code>	checks if an image is writable, no read-only flag.
<code>clut_ok</code>	checks if the pointer supplied is a valid color lookup table
<code>pl_io_ok</code>	checks if the specified bitplanes are in the correct range
<code>plane_ok</code>	checks if the plane is in the correct range
<code>im_val_ok</code>	checks if value(s) is(are) smaller than the image size(s)
<code>val_check</code>	checks if a value is smaller than the image size
<code>iter_ok</code>	checks if the number of iterations is positive or zero
<code>edge_ok</code>	checks if the edge bit parameter is correct
<code>con_ok</code>	checks if the connectivity parameter is correct
<code>con6_ok</code>	checks connectivity parameter on 4, 8, 48 or 84
<code>odd_fsizes_ok</code>	checks if filter sizes are odd and in the specified range
<code>range_ok</code>	checks is a value is in the specified range
<code>frange_ok</code>	checks is a float value is in the specified range
<code>odd_ok</code>	checks if a value is a odd integer value
<code>even_ok</code>	checks if a value is a even integer value
<code>bit_ok</code>	checks if a value is a binary value
<code>different_ok</code>	checks if two values differ from each other
<code>positive_ok</code>	checks if an integer value is positive
<code>fpositive_ok</code>	checks if a float value is positive
<code>greater0_ok</code>	checks if a integer value is bigger than zero
<code>fgreater0_ok</code>	checks if a float value is bigger than zero
<code>unequal0_ok</code>	checks to see if an integer value is unequal to zero
<code>funequal0_ok</code>	checks to see if a float value is unequal to zero
<code>power_of_2_ok</code>	checks if a value is a power of 2

Please refer to the (on-line) manual pages for a complete description of these functions.

Apart from these checking functions that generate an error, also some testing functions exist that only return a true/false value and not generate an error. These functions all start with the "is_" prefix, they are:

<code>is_image</code>	checks if the pointer supplied is a valid image pointer
<code>is_clut</code>	checks if the pointer supplied is a valid color lookup table

Check_image_integrity

The function `check_image_integrity()` checks all images for irregularities. For instance, if the input and output descriptor of an image are different, it will delete the image connected to the output descriptor and reconnect `out_descript` to the same descriptor as `in_descript` (see also "Figure 9-1" and "Figure 9-2"). Note that this is exactly the opposite of what `post_op()` does. Therefore this function should not be called before a call to `post_op()`, unless the intention is to restore the state of affairs prior to the call to `pre_op()` and exit the function immediately. The function heading is:

```
void check_image_integrity( int print )
```

The argument of `check_image_integrity()` tells the function to perform its work silently (0) or print what it is doing (1). This function is called by the Image infrastructure just before it publishes the occurrence of an error.

Textual output

Because the Image Library is designed to be independent of a user-interface, text-output cannot be written to a "standard-output stream" because that may not exist. Also the User-interface might like to be in control of the representation of textual output. For these reasons, Image functions should not use the standard `printf(...)` and `fprintf(...)` functions for text-output to a "terminal" (the `fprintf()` may still be used for specific file output of course).

The Image library contains a special function for text-output, the `image_output()` function. The prototype:

```
int image_output( int stream, const char *format, ...);
```

The syntax of the function and the parameter list is like the ANSI-C `fprintf()` function, except that the "stream" parameter does NOT indicate a C-FILE streams. It merely defines the type of the text and should be considered an aid for the user-interface to enable it to present different kinds of text in different ways. The default behavior of `image_output()` is to dump all text to `stdout` except for the `IMO_ERROR` stream which is dumped to `stderr`.

User interfaces can intercept the text by supplying a function (pointer) that conforms to the following prototype.

```
void WINAPI1 funcname( int stream, char *buffer);
```

The `image_output()` function converts its variable argument list to a single text buffer that is given to this function. To supply the function-pointer, use the function

```
#include "imtxtout.h"
```

```
im_set_output_handler(void (WINAPI *) (int, char *));
```

The defined types of text are (definition can be found in the include file 'image.h'):

¹ We use WINAPI on the MS-Windows platform to provide greater flexibility. On other platforms WINAPI is an empty define (see the include file `imtxtout.h`)

<i>stream</i>	<i>intended type of text</i>
IMO_OUTPUT	default text such as measurement results, progress information etc.
IMO_INSTRUCT	instructions for the users, e.g. information on how to operate an interactive routine.
IMO_WARNING	warning messages
IMO_ERROR	error messages

Table 9-3 : present output streams.

Please note that although we define different ‘streams’ of text-output, we do not impose any behavior on these streams. It is up to the user-interface to present the text in a suitable layout.

Function overloading

Image uses the mechanism of function overloading for flexibility, maintainability and simplicity. The mechanism works by having the top-level image-processing functions look at the type of the image and calling a low-level functions that is tailored for that type of image.

Here we consider how that mechanism works and is implemented in Image. With use of the function overloading mechanism, new image processing routines can be implemented for several types of images or existing operations can be implemented for additional image types.

Three layers

In order to make use of the function overloading the source code should be divided into three layers:

- 1) generic function level
- 2) parameter checking and type and size adjustment
- 3) processing of the data

By splitting the processing of the image in layers 2 and 3, the actual processing function (layer 3) can be used for more than one image type. If the image adjustments and parameter checking (layer 2) differ per image type, this is especially useful.

Consider the function `add_im()` as an example to describe the three layers. `add_im()` adds the values at corresponding locations in two images and stores the result in another image. The call to `add_im` is `add_im(A, B, C)`. `A` and `B` are the input images and the result is stored in `C`. The images `A` and `B` are `GREY_2D` (in this example) and since `C` is an output image, it is converted to `GREY_2D` with `pre_op()` as described in Section "Dynamic adjustment (Pre_op, Post_op)" on page 9-12.

Generic function layer

The function specified as the generic function is called by the user/application. It calls the function-overloader to acquire the appropriate function for the image type, and then calls that function with its arguments. For the `add_im()` operation, the generic function is:

```
/* necessary structures, defines and prototypes */
#include <stdlib.h>
#include "image.h"
#include "im_error.h"
#include "generic.h"
#include "im_infra.h"

int add_im( IMAGE *in1, IMAGE *in2, IMAGE *out)
{
    int status;
    int (*func)(IMAGE *, IMAGE *, IMAGE *); /* pointer to function */

    im_begin_func("add_im");

    /* find function pointer for type of 'in1' */
    func = overload_func("add_im", in1 );

    /* if not present for this image type, report the error */
    if (!func)
        return( im_report_error("add_im", IE_NOTOVERL, ""));

    status = (*func)(in1, in2, out); /* perform operation */

    /* if error detected, exit function with error-value */
    if (status<IE_OK)
        return(im_report_error("add_im", status, ""));

    /* normal exit */
    im_end_func("add_im");
    return(IE_OK);
}
```

The function `overload_func()` returns a pointer to the function that performs the add operation for the type of the `in1` image. In the tables that contain the overloadable functions for each image type, `add_im` is an entry and is accompanied by a function-pointer to a type-specific function. In the table for `GREY_2D` images this pointer points to the function `g_add()` and, as a result of that, this function will be called:

```
g_add( in1, in2, out);
```

Parameter checking and image adjustment

The function `g_add()` is the second layer:

```
#include "image.h"
#include "im_error.h"

int g_add( IMAGE *in1, IMAGE *in2, IMAGE *out )
{
    im_begin_func("g_add");
```

```
if (!pre_op(in1,in2,COMPARE,G_2D_SPEC|G_3D_SPEC,OUT_AS_IN) ||
    !pre_op(in1,out,ADJUST,G_2D_SPEC|G_3D_SPEC,OUT_AS_IN))
    return( im_report_error("g_add", IE_PRE_OP,""));

l_g_add( in1, in2, out );
if ( (status = post_op(out)) < IE_OK )
    return(im_report_error("g_add", status,""));

im_end_func("g_add");
return(IE_OK);
}
```

This second layer is used for image type and size adjustment as well as general parameter checking. The functions `pre_op()` and `post_op()` which test the input images and adjust the output image are discussed in detail in "Dynamic adjustment (Pre_op, Post_op)". Here the first call to `pre_op()` compares the images `in2` and `in1` and produces a warning if either the dimensions or the type differ. The second call to `pre_op()` will adjust the image `out` to be of the same dimensions and type as image `in1`. If all checks and adjustments succeeded, the low-level function that does the actual computing is called (`l_g_add()`). When `l_g_add()` completes its task, the result is displayed, with the `post_op()` function. Because `add_im` is a point operation, the same `add` routine (`l_g_add()`) can be used for both two and three dimensional images. Because neighborhood operations are always dimension dependent, different low level routines have to be created.

Processing the data

The function `l_g_add()` (low level grey value add) is the function which performs the actual calculations on the image data:

```
#include "image.h"
#include "im_error.h"

void l_g_add(IMAGE *in1, IMAGE *in2, IMAGE *out )
{
    register PIXEL      *src1, *src2, *dst;
    register int        npix;

    im_begin_func("l_g_add");

    src1 = ImageInData(in1); /* address of input data of 'in1' */
    src2 = ImageInData(in2); /* address of input data of 'in2' */
    dst = ImageOutData(out); /* address of output data of 'out' */
    npix = ImageSize(in1);   /* number of pixels in image 'in1' */

    while( --npix >= 0 )
        *dst++ = *src1++ + *src2++;

    im_end_func("l_g_add");
    return;
}
```

The type of data in grey valued images is `PIXEL`. The addresses of the data in the images and the number of pixels in the image are retrieved using macros that reside in the include-file

"image.h". Since nothing can fail while when adding (all checking has been done in `g_add()` and overflow is unlikely since each pixel is 16 bits) we do not return a status.

Image supplies several example files to illustrate the mechanism of overloading. These files can be found in the "src_exmp" directory. In that location directories are located for a few image types. In each of these directories source files for the corresponding image types are located. In the "grey_2d" directory, the files "arith.c" and "arithlow.c" contain the functions `g_add()` and `l_g_add()`. Compare these functions with the functions `f_add()` and `l_f_add()` ("float_2d" directory) to see the difference in how grey valued images and floating point images are handled. Further, several examples of generic level functions are found in the "generic" directory.

Overload tables

Image determines which function it must call to perform a certain operation on an image type by means of **overload tables**. The programmer specifies the type-specific functions in a plain text file, the 'overload files'. These files can be recognized by the '.ovl' file-extension. At compile-time a special utility program (**mkoverld**) converts these overload files to a C-source code file (overload.c) containing several arrays. This C-file is then compiled and linked with the Image library.

The layout of the overload files is not very complicated and looks like this:

```
#
# optional comment lines describing the contents
#
Definition of the image type this table is for.
    message-functions pair definition (see below)
    message-functions pair definition
    message-functions pair definition
    message-functions pair definition
    ...
```

The rules defining this layout are:

- All lines that start with a pound sign (#) are comments. Empty lines are also regarded as comment.
- The first non-comment line found in the file must be a table definition using the following syntax:

```
TABLE <window name> <IM IDENT> <IM SPEC> <IM DIMENSIONS>
```

<window name> one of : g2d, g3d, f2d, f3d, b2d, b3d, c2d, c3d, col2d, col3d, l2d, l3d.

<IM IDENT> is the type identifier for this image type.

<IM SPEC> is the type specifier for the image type.

<IM DIMENSIONS> is the number of dimensions the image type has.

The standard table for the GREY_2D image type is specified by:

```
TABLE g2d GREY_2D G_2D_SPEC 2
```

- The lines in the file following the line with the TABLE keyword are *message-function* pairs. The first word on the line is the name of the operation (the generic function layer) and the second word is the name of the function that performs that operation for the image type involved. When 'overload.c' is created the first word is converted to a string and the second one into a function address. For the **add_im** operation, the 'message-function' pair for image type GREY_2D is:

```
add_im g_add
```

- Every time that the keyword **TABLE** is encountered in an overload file, a new table will be created. This way, multiple tables can be specified in one file.
- When the 'overload.c' is created it is not checked whether a name of an operation is already in use. If that happens the last occurrence of the name (and the corresponding function) overrules the previous one(s).

Note that in the overload file for GREY_3D images, the message-function pair for the add_im operation is the same as the one in the overload file for GREY_2D images. This is because the g_add() function can be used for both GREY_2D and GREY_3D images.

Overruling the default implementation

It can occur that a user of Image is not satisfied with the implementation of an image processing operation or simply wants to experiment with a different implementation. For instance, if the default implementation of a filter for GREY_2D images is not suited for the specific images used, the Image routine can be overruled by a different one. This can only be done if the new function is called with the same set of parameters as the old one, because the generic level function is unchanged. If the parameters differ, a complete new function must be implemented using a different name.

To overrule the Image implementation of a function, the 'message-function' pair (rule) for the new function must be located later than the rule for the default implementation or, off course, replace the existing rule. This can be done by placing it later in the same file as a pair with the same message, or by placing it in a new file which is lower in the list of overload files from which 'overload.c' is created. The location of this list depends on the platform you are using Image on, it is either in the "makefile" (UNIX & PC) or in the file "ovl_list" (Macintosh).

Data conversion (convert)

Occasionally, conversion of the image data is required, for example if an operation on grey valued image data must be performed in floating point precision. A special mechanism is supplied which enables one function to convert any type of image in Image into any other type. Even if new image types are implemented, no altering of existing source code is necessary. Because the `convert()` command makes use of the function overloading mechanism, users can overrule the method used to convert the image data.

Super type of an image line

The routine `convert()` uses the special image line type, called a *common_line* to store a line of image data. By having a routine for each image type that puts the data of an image line in a `common_line` structure, and a routine for each image type that reads a line from the `common_line` structure, any type of image data can be converted to any other. The entire image is converted by calling the former function with the input data, and the latter with the output image data, for each line in the image.

The data in the `common_line` is stored either in long words or in doubles. The choice of which, is determined by the routine that stores a line of the source image into the `common_line`. The receiving routine of the destination image must therefore be able to read the data from the long word format and from the double format. The receiving routine does the actual conversion. To summarize, every image thus needs two routines for this conversion:

- One to store the data of the image on a line by line basis into a `common_line` structure either in long words or in doubles .
- One to read the data from the `common_line` and store it in the output image. This routine must be able to read the data in the `common_line` structure, which means that it must be able to read both long words and doubles.

COMMON_LINE structure

The `common_line` structure (`COMMON_LINE`) is defined in the file 'image.h' like this :

```
#define COM_LONG    -21
#define COM_DOUBLE -22

typedef struct common_line_t {
    int          type;          /* type of data */
    void         *data;        /* pointer to the data */
    unsigned int x, y, z;      /* dimensions and position line */
    unsigned int t;           /* time position of the line */
    unsigned int nr_channel;   /* number of channels per pixel */
    double       hint_min;     /* minimum value in data */
}
```

```
double      hint_max;    /* maximum value in data */
unsigned int type_ident; /* ident of data source */
unsigned long type_spec; /* spec of data source */
} COMMON_LINE;
```

- `type` identifies the C type in which the data is stored, `COM_LONG` is the long word format and `COM_DOUBLE` is the double format.
- `*data` is a pointer to the allocated memory used to store the line of data (see "Source function specification" on page 9-33).
- `x` is the width of the image
- `y` and `z` are the position of the line stored in the data.
- `t` is the time position of the line (not used in current image types).
- `nr_channel` is the number of values used for one pixel. For example, grey value images are one channel images and complex images are two channel images.
- `hint_min` and `hint_max` can be used to store the minimum and maximum value of the 'source' image. This information can be useful, in deciding how to convert the data in the receiving routine.
- `type_ident` and `type_spec` are the `type_ident` and `type_spec` of the source image.

When `convert()` is called the functions for writing to a `common_line` for the source image and reading from a `common_line` for the destination image are overloaded. Each is called several times until the image is completely converted. Each function must have a specific behavior to guarantee the correct conversion of the data.

Source function specification

The header of the source function is specified as follows:

```
int source_function( IMAGE *image, int n, COMMON_LINE *com_line)
```

- `image` is the input image.
- `n` is number times the function has been called (when first called, `n = 0`).
- `com_line` is a pointer to the `COMMON_LINE` structure used in the **convert** operation.

- The first time the source function is called ($n = 0$), it must allocate memory to hold a line of data from the source image and fill the structure `COMMON_LINE` to describe the format of the data. To fill the structure, the function `set_common_line()` can be used (see on-line manual). The allocated memory is freed by the function `convert()` when the conversion is completed.
- The fields `y` and `z` are filled each time the source function is called with the position of the line in the source image. In a 2 dimensional image, $y = n$. In 3D images `y` starts from zero each time the `z`-position is raised. `z` also starts from zero and ends at the value (`depth - 1`).
- The fields `hint_min` and `hint_max` must be set to a value, either the exact minimum and maximum value in the image or a good estimate thereof. In the Image implementation, these values are set to 0.0 and 255.0 for all non binary image. For binary images, the values are set to 0.0 and 1.0.
- Every time called, the source function fills the memory with the next x-line of the source image.
- The return value of the function is used to indicate whether more lines follow the present line. If there are lines in the image that have not yet been transferred, the function should return the value '1' (or any other non-zero value). When the last line is processed, the function returns the value '0' (zero).

The source of the default grey value function is listed below for reference :

```
int g_2d_conv_to_common(IMAGE *im, int n, COMMON_LINE *com_line)
{
    GREY_2D_IMAGE *g_im;
    PIXEL *pix;
    long *data;
    register long npix;

    im_begin_func("g_2d_conv_to_common");

    g_im = (GREY_2D_IMAGE *) ImageIn(im);
    npix = Grey2dImageWidth(g_im);

    if( n == 0 ){          /* First time called */
        data = (long *)malloc((size_t) npix * sizeof(long));
        if(!data)
            return(im_report_error("g_2d_conv_to_common",
                                   IE_NOMEM, "no memory allocated for data"));
        set_common_line(com_line, COM_LONG, data, 0, 0, 0, 0, 1,
                        0.0, 255.0 );
    }

    data = com_line->data;
    com_line->x = npix;
    com_line->y = n;
    com_line->z = 0;

    pix = Grey2dImageData(g_im);
    pix += com_line->y * npix;
```

```
while(--npix >= 0 )
    *data++ = *pix++;

if( n == Grey2dImageHeight(g_im) - 1 ){
    im_end_func("g_2d_conv_to_common");
    return( 0 );
    /* 0 = stop, this is the last line */
}

im_end_func("g_2d_conv_to_common");
return( 1 ); /* 1 = more lines are available */
}
```

Destination function specification

The header of the destination function is specified as follows:

```
int destination_function( IMAGE *image, int n, COMMON_LINE
                        *com_line;
```

- `image` is the destination image.
- `n` is number of times to the function has been called (when first called, `n=0`).
- `com_line` is a pointer to the `COMMON_LINE` structure used in the convert operation.
- The destination function is called after each call to the source function and must read the data from the `common_line` and put it in the image at the position specified by `y` and `z`. If a 3D image is converted into a 2D image then the function may stop if the `z` value is more than zero or take another appropriate action. In the default implementation, for example, the values in other `z` positions than 0 are compared with the values already stored in the 2D image and the maximum is taken as the value for that pixel. This method provides a good 2D representation of most 3D images.
- The destination function must be able to read the data in the `common_line` regardless of whether it is in long word format or in double format. The fields `hint_min` and `hint_max` can be useful for the conversion.
- The return value of the function should indicate if the function can handle more lines of data. If more data can be processed, a non-zero value should be returned. If, for some reason, the destination function cannot continue processing it should return a 0 (zero) to stop the conversion.

The source of the default grey value function is listed below for reference :

```
int g_2d_conv_from_common(IMAGE *im, int n, COMMON_LINE *com_line)
{
    GREY_2D_IMAGE    *g_im;
    double           *d_ptr;
    long             *l_ptr;
```

```
PIXEL          *pix;
int            type;
register long  npix;
register int   chan_offs;

im_begin_func("g_2d_conv_from_common");

g_im = (GREY_2D_IMAGE *) ImageOut(im);

type = com_line->type;
d_ptr = com_line->data;
l_ptr = com_line->data;

chan_offs = com_line->nr_channel;

npix = Grey2dImageWidth(g_im);
pix = Grey2dImageData( g_im );
pix += com_line->y * npix;

if( com_line->z == 0 ){
    if( type == COM_LONG ){
        while( --npix >= 0 ){
            *pix++ = *l_ptr;
            l_ptr += chan_offs;
        }
    }
    else if( type == COM_DOUBLE ){
        while( --npix >= 0 ){
            *pix++ = *d_ptr;
            d_ptr += chan_offs;
        }
    }
}
else {
    if( type == COM_LONG ){
        while( --npix >= 0 ){
            if( *pix < *l_ptr )
                *pix = *l_ptr;
            pix++; l_ptr += chan_offs;
        }
    }
    else if( type == COM_DOUBLE ){
        while( --npix >= 0 ){
            if( *pix < *d_ptr )
                *pix = *d_ptr;
            pix++; d_ptr += chan_offs;
        }
    }
}

if( com_line->t > 0 ){
    im_report_error("g_2d_conv_from_common", IE_NOT_OK, "");
    return( 0 );
    /* 0 = stop, can't handle this */
}

im_end_func("g_2d_conv_from_common");
return( 1 ); /* 1 = can handle more data */
}
```

User specified conversion

The routines that perform the actual conversion of the data are overloaded for each image type. In "Overruling the default implementation " on page 9-31, we explained that overloaded functions can be overruled by the user. By overruling the type specific functions for the convert operation, the user can implement different conversion methods for certain image types. All that needs to be done is to write a function that meets the specifications for the source or destination function described above. The new function can overrule the default function for convert by overloading the user's function for the messages `conv_to_common()` (for the source function of an image type) or `conv_from_common()` (for the destination function of an image type). In the standard overload files these messages and the default functions can be found just below the obligated service functions.

Var_objects

Var_objects in Image are used to store non-image data which is handled by the user. They prevent the user from having to manage data arrays and addresses thereof in the interpreter or dialogue boxes. To simplify the use of var_objects, they are handled like images. That is, when results are stored in them, they can be adjusted dynamically to accommodate the data. Because the var_objects can be used to store various types of data, a simple mechanism is provided to determine which appear in a dialogue box. This enables the (application) programmer to restrict the visible var_objects to those the user may need to manipulate.

Var_object structure

The structure that describes a var_object is found in the include file 'image.h' :

```
typedef struct var_object_t {
    int     type;                /* type of data in this object */
    void    *data;              /* pointer to the data */
    int     nr_channel;         /* number of channels per element */
    int     dimensions;        /* number of dimensions */
    int     dims[V_O_MAX_DIM]; /* dimensions */
    int     size;              /* total number of elements */
    int     bpelem;            /* bytes per element */
    char    object_name[IM_NAMELEN]; /* name of the object */
    char    object_class[IM_NAMELEN]; /* class of the object */
    char    *comment;          /* pointer to possible comment */
} VAR_OBJECT;
```

As with the IMAGE structure, all members of this structure can be referenced with macros:

```
#define V_O_Type(op)          (op)->type
#define V_O_Data(op)         (op)->data
#define V_O_Chans(op)        (op)->nr_channel
#define V_O_Dims(op)         (op)->dimensions
```

```
#define V_O_Dim(op, dimension)      (op)->dims[dimension-1]
#define V_O_Size(op)                (op)->size
#define V_O_ElemSize(op)            (op)->bpelem
#define V_O_Name(op)                (op)->object_name
#define V_O_Class(op)               (op)->object_class
#define V_O_Comment(op)             (op)->comment

/* for the total number of bytes in the object */
#define V_O_Length(op) (V_O_Chans(op)*V_O_Size(op)*V_O_ElemSize(op))
```

- `type` is the type of the data the `var_object` contains, it can have one of these values: `PIXEL_T`, `CHAR_T`, `SHORT_T`, `INT_T`, `LONG_T`, `FLOAT_T`, `DOUBLE_T`
- `data` is the pointer to the memory that holds the data stored in the `var_object`.
- `nr_channel` is a special dimension, introduced to support data types with multiples of data, such as like complex values, more than one value is stored in each unit of the object.
- `dimensions` is the number of dimensions in the `var_object`. It can range from 0 to `V_O_MAX_DIM`, which is 5 at present. This is equivalent to the number of dimensions in an array. By using 0 dimensions, a scalar can be stored in a `var_object`.
- `dims` is an array that contains the length in each dimension in the `var_object`.
- `size` is the total number of elements in the `var_object`. It is the product of all the values in `dims` array. The number of channels is not included in the product.
- `bpelem` stands for bytes per element and contains the number of bytes used in each element of the data. The number of channels per element is not included. If a `var_object`, for instance, has four channels (`nr_channel == 4`) and the type is `char` (1 byte) then `bpelem` is also 1.
- `object_name` is a string that contains the name of the `var_object`. This name is used in the dialogue boxes and in the command window. The maximum length of a `var_object` name is `IM_NAMELEN` (see "image.h") characters.
- `object_class` is the object class of the `var_object` belongs to. In the comfile, this name is specified in the minimum field. Only `var_objects` that have that name in the `object_class` string will appear in the dialogue box.
- `comment` is a pointer to a string of arbitrary length. This string can be used to store a comment concerning the `var_object`. When the `var_object` is written to disk, the comment is saved in the header file of the `var_object`.

The macros should be self-explanatory, but a note on the `V_O_Length` macro may be in place. It is used to determine the size of the entire `var_object` in bytes. To be specific, it is the product of the fields `nr_channel`, `size` and `bpelem`.

Programming with var_objects

The var_objects are designed to be an easy interface to C arrays for the user. Any operation that deals with arrays of data that must be accessible to the user, should use var_objects. The var_objects created are stored in a linked list. When programming a function, the var_objects can be used like normal arrays once they have been created. Low-level functions however that need arrays, may also use var_objects rather than arrays. The advantage is that their size and type can be easily adjusted.

To dynamically adjust the type and the sizes of var_object, several functions are available:

var_object	creates a var_object and returns the pointer
destroy_var_object	destroys a var_object
var_object_by_name	retrieves the pointer to a var_object through its name
set_var_object_type	sets the type of a var_object to a specific type
set_var_object_size	sets the dimensions of a var_object to specific values
set_var_object_data	sets both the type and the sizes of a var_object
set_var_object_class	sets the class of a var_object
set_var_object_comment	attaches comment to a var_object.

Several useful functions are available for inspecting and manipulating the var_object as a whole:

list_var_objects	lists all var_objects that exist
show_var_object_info	shows information on a var_object
dump_var_object	dumps the contents to the terminal or a file
copy_var_object	copies a var_object to another one
write_var_object	saves a var_object to disk
read_var_object	reads a var_object from disk

These functions have a (on-line) manual page that contains information on them.

Checks on var_objects

The function `is_var_object()` can be used to determine the existence of a var_object. It returns the value '1' if the var_object exists, and '0' otherwise. The function `var_object_ok()` does the same, and pops up an alert box if the var_object does not exist.

Conversion of var_objects to images and vice versa

Var_objects are implemented to simplify the management of non-image data for the user. As a var_object may contain data that could also be stored in an image, it might be useful to process the data in a var_object with one of the image processing functions. To make this

possible, conversion routines have been written to transfer the data from a `var_object` to an image and vice versa. A routine is also provided to convert the data in a `var_object` to another type. The routines are :

<code>var_object_convert</code>	changes the data into another type
<code>var_object_to_image</code>	transfers the data to an image
<code>image_to_var_object</code>	stores data from an image in a <code>var_object</code>

A `var_object` is converted in the same way as an image. A `common_line` is used to store one line of data, which is then read by the receiving object (see "Data conversion (convert)" on page 9-32). In fact the destination functions that are used to transport the data between the `common_line` and an image are the same functions that are used for image conversion. This means that if the source or destination function for an image type is overruled by the user, the conversion of `var_objects` to or from images is also overruled.

The source and destination functions which handle the transport between `var_objects` and the `common_line` however, cannot be overruled. The data contained in an image, regardless of its size or type, always fits in a `var_object` without loss of information .

Histogram objects

The histogram data of images in Image are stored in histogram objects. These histogram objects can be of any size and have up to 5 dimensions. For each dimension of the histogram, the center pixel value of the first and the last cell are stored as well as the width of one cell. Functions are available to create, destroy and resize the histogram objects.

Histogram structure

The definition of the HISTOGRAM structure, located in the include file 'image.h', is :

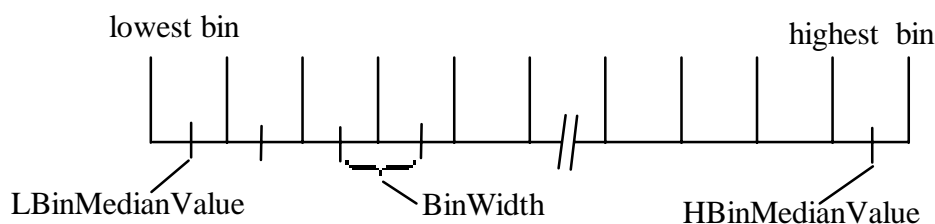
```
typedef struct histogram_t {
    unsigned long *hdata;
    unsigned int  nr_chans;
    unsigned int  nr_dims;
    unsigned int  dims[V_O_MAX_DIM];
    char          name[IM_NAMELEN];
    char          *comment;
    double        lbin_median[V_O_MAX_DIM];
    double        hbin_median[V_O_MAX_DIM];
    double        bin_width[V_O_MAX_DIM];
    void          *fut1; /* reserved */
    void          *fut2; /*   for   */
    void          *fut3; /* future  */
    void          *fut4; /*   use   */
} HISTOGRAM;
```

For all used fields in the structure, a macro has been made for accessing the field.

```

HistogramData (hp)                (hp)->hdata
HistogramChans (hp)                (hp)->nr_chans
HistogramDims (hp)                 (hp)->nr_dims
HistogramDim (hp, dimension)       (hp)->dims[dimension-1]
HistogramLBinMedian (hp, dimension) (hp)->lbin_median[dimension-1]
HistogramHBinMedian (hp, dimension) (hp)->hbin_median[dimension-1]
HistogramBinWidth (hp, dimension)  (hp)->bin_width[dimension-1]
HistogramName (hp)                 (hp)->name
HistogramComment (hp)              (hp)->comment
    
```

- `hdata` is the pointer to the actual histogram data. The type of the data is unsigned long.
- `nr_chans` is the number of 'channels' of the histogram data. When the image data is 'multi-channel' data, like RGB color-images, the histogram can be stored in three channels to get an overview of how each of the three colors are distributed.
- `nr_dims` is the number of dimensions of the histogram. The histogram objects can handle up to `V_O_MAX_DIM` (5) dimensions.
- `dim` is an array of `V_O_MAX_DIM` (5) unsigned int's in which the dimensions are stored.
- `name` is a array of `IM_NAMELEN` (80) char's in which the name of the histogram objects is stored. This name is used by the dialog boxes to identify the histogram objects.
- `comment` is a pointer to a string of arbitrary length. A comment string of any length can be attached to the structure using the `histogram_comment()` function. This function allocates memory for the string and frees the memory of an previously added comment string.
- `lbin_median` is a double that describes the center pixel value of the lowest cell of each dimension in the histogram, see example below.
- `hbin_median` is a double that describes the center pixel value of the highest cell in the histogram, see figure below.
- `bin_width` is a double that describes the width of the cells of the histogram, see figure below.



The fields `lbin_median`, `hbin_median` and `bin_width` describe which pixel value of the image is stored in which cell of the histogram. The width of each cell, already calculated and stored in the `bin_width` array, can be calculated by the formula :

$$bin_width = \frac{hbin_median - lbin_median}{\#bins - 1}$$

The center pixel value of each cell n of the histogram can be calculated through :

$$center_value_n = lbin_median + n \cdot bin_width$$

In which n is the number of the cell (starting from 0). To determine in which cell certain pixel value would be stored, the following formulas are used:

$$low_border = lbin_median - \frac{bin_width}{2.0}$$

$$bin = \frac{pixelvalue - low_border}{bin_width}$$

For instance, the histogram of grey-valued integer image data ranging from 0 to 255 and stored in a histogram of 256 cells would show the following values. `lbin_median` is 0 (the lowest value of the image data), `hbin_median` is 255 (the highest value of the image data) and `bin_width` is 1 (each cell is exactly 1 pixel value wide).

Consider the histogram of floating-point data ranging from 0.0 to 1.0 and stored in 100 bins. The `lbin_median` is 0.0, the `hbin_median` 1.0 and the `bin_width` $1.0/99 = 0.0101$. The center pixel value of bin #62 would be 0.6262 (= 0.0 + 62 · 0.0101).

The `histo_data()` function calculates the histograms of all image types in Image (except for complex images). This function is capable of calculating the histogram of sub-ranges of the image data. When such a histogram is calculated, the above formulas remain valid but after the bin has been calculated, it must be checked if it is within a valid range (0 .. number of bins).

Programming with histogram objects

Several functions are available to create, destroy and view the histogram objects. They are :

<code>create_histogram</code>	creates an empty histogram object.
<code>destroy_histogram</code>	destroys a histogram object.
<code>histo_data</code>	gets the histogram data from an image.
<code>dump_histogram</code>	show the histogram data in ASCII.
<code>show_histogram_info</code>	show textual information of a histogram.

When using the histogram objects, you may need some additional functions for inspecting and manipulating the histogram objects:

histogram_comment	attaches a comment string to a histogram object.
histogram_by_name	gets the pointer to a histogram objects using its name.
is_histogram	checks if a pointer is a valid, existing histogram object and returns true or false.
histogram_ok	checks if a pointer is a valid, existing histogram object and pops an alert box if this is not.
list_histograms	lists all existing histogram objects
copy_histogram	copies the data of one histogram to another

When it is necessary to copy the data in the histogram object to an image or var_object or vice versa, these functions can be used.

histogram_to_image	copies the histogram data to an image.
image_to_histogram	copies image data to an histogram object.
histogram_to_var_object	copies the histogram data to a var_object.

More detailed information on the functions listed above can be found in the reference pages for these functions.

Chapter 10 Analysis of Images and Objects (AIO)

This chapter describes the library for Analysis of Images and Objects (AIO). AIO is an infrastructure for interactive and automatic object measurements. It provides mechanisms for object selection, and for storage of objects in image silos. The general concept is discussed and examples of the use of AIO are presented.

Do not read this chapter:

- If you are a beginning user.
- If you have never programmed in C.

You should read this chapter:

- If you are an experienced user with a basic knowledge of C programming.
- If you want to perform object measurements.

Introduction

AIO is a framework for the analysis of objects in images. It is especially suitable for performing measurements on (microscopical) objects. Objects are manipulated either on the basis of the measurements or by pointing at their images. The object data vector and the object image can be stored in a file and in an image silo respectively, for later retrieval. The combination of AIO and Image is a flexible environment for image analysis, and application development.

General Concepts in Microscopical Image Analysis

Most microscopical applications measure certain features of objects in images, and/or classify objects into different categories and therefore adhere to this general scheme:

**Acquisition -> Restoration -> Segmentation -> Identification ->
Feature extraction -> (Selection -> Classification)**

In a number of steps, depending on the nature of the problem at hand, images are segmented into a binary image with object pixels set to '1' and background pixels set to '0'. Component labeling, that is assigning an identifying label to a set of connected pixels belonging to the same object, results in an identified or labeled image. The labeled image is either used to delineate shape features, or it is used as a mask while measuring grey value features of objects in the original image. Because the same scheme is generally used in many different applications, a flexible and generalized framework can improve application development substantially.

Strikingly enough, in many image processing systems, all measurements are applied to all the objects in the image and a specification of all requested features needs to be given prior to the analysis. The measurements of all objects are then carried out in a single pass through the image, and results are printed on the terminal or dumped to a file for further evaluation.

With this approach, it is cumbersome to access and use the measurement results in a program to select or further manipulate objects. In addition, the single pass approach has a major disadvantage when employing hierarchical classifiers, especially in time critical applications. When the time needed for the measurement is a point of concern, it is better to apply a hierarchical classifier which (depending on the state of a decision tree) sequentially decides what feature of an object to measure next. We therefore decided to support the measurement of individual objects rather than the measurement of all the objects in the image.

In addition to the single object measurement, the manner in which objects are stored is important for microscopical image analysis applications. For instance, in automatic screening for medical applications, requests to store images of measured or classified objects for later inspection by a medical expert are common. When building a classifier, stored object images may be used to construct test and learn sets. How the images are stored plays a role in the access times for these operations.

Another important consideration in microscopical image analysis is the style of user interaction. In many applications the capacity to interactively manipulate the objects, adds to the usability and functionality of the application. It may be useful, for example, to point at an object of special interest to initiate a measurement, or to correct for mis-classifications. We therefore support both automatic and interactive microscopical image analysis.

Based on the preceding discussion, the AIO framework is designed to support microscopical image analysis with a focus on objects, and attention to the storage method for images of individual objects, and the human interface.

Components of the AIO framework

The AIO package is completely based on linked list manipulations and computations on individual objects. The framework consists of various parts, which we now consider briefly.

Labeling objects

During the image labeling process, a data structure is assigned to each image object to store information and measurement results as they become available. To allow multiple objects in an image to be viewed as a unit, and to support image measurement, a linked list of objects is built with information regarding each object's dimensions and positions.

Measuring individual objects

For feature extraction, each object is passed to a measuring routine which calculates the specified features, storing the result in a feature/value pair connected to the object data structure. Measurements are carried out in sub-images, processing only pixels in the rectangle enclosing the object. The time needed for measurement is therefore decreased substantially. Further, because the features need not be specified simultaneously, a pre-selection based on simple operations can be made for object classification. The fact that features to be measured need not be specified simultaneously is highly useful for purposes of object classification. A

pre-selection can be made based on fast measurements. The time-consuming measurements need only be applied to those objects which survived, lowering the classification costs.

Object manipulation

To manipulate the objects, basic utility functions are provided. They are removing an object from an image, copying an object from one image into a randomly positioned window of another. As objects are referred to through their associated data structure, measured results can easily be accessed. A feature can be measured on every object by applying the measurement to each of the objects in the list. A selection criteria may be applied while traversing the list.

Image Silo

An image-silo package enables objects or group of objects to be stored in image silos, for later retrieval and possibly further analysis. Furthermore, for presentation purposes, routines are provided for laying out groups of objects in the form of composite images.

Direct manipulation

Event driven interaction based on mouse and keyboard input is supported in AIO¹. AIO enables simple interactive selection of objects by scanning the object list and returning the object at a given location. Direct manipulation and user interfaces for image object manipulation can be written based on it.

An AIO sample session

In a number of examples some of the concepts of AIO are demonstrated. Although this the code fragments in this section can be used in a stand-alone program without image-display, it is advised to execute this session in the SCIL_Image package as this shows the result of each statement directly on the screen.

First an image must be obtained. In our example a file is read from disk, thresholded, and inverted so that object pixels are set and background pixels are cleared :

readfile(cermet, A, 0, 0);	Obtain an image with objects
threshold(A, B, 128);	Segment the image

¹Not present in the stand-alone Image Library (see "Interactive measurement" on page 10-6).

invert_im(B, B); Object pixels set, background zero

Given a segmented image with objects and background the image must be labeled, returning a list with objects. In order to use AIO, data structures must be defined, using the standard C file inclusion mechanism :

```
#include "im_aio.h"           AIO data structure definitions
LIST *o_list, *object;       Two list variables
o_list = list_label(B, C, 8, 0); Label the image, return a list
```

Since the labeling process creates a list with initial information, including the area of the objects, objects too small or too big to be of interest can be eliminated immediately. In our example, objects touching the edge of image C are removed as well. Removal from the list is sufficient to never reference the objects again, but for clarity, we choose to erase the objects from the image as well :

```
FORALL(object, o_list)
    if (edge_object(C, object) || area(object)<20 || area(object)>400) {
        hide_object( C, object );    Erase object from image
        rm_object( object );         Mark object for removal from list
    }
o_list = update( o_list );          Update the object list
```

Next the optical density of the remaining objects is measured :

```
FORALL(object, o_list) object_dens_meas(A, C, object, OD);
```

To store the grey value image of the objects in image A with a mean optical density lower than 50, an image silo must be created :

```
long silo;
silo = create_silo( "test.silo" );
int object_number = 0;

FORALL( object, o_list )
    if( od_mean(object) < 50 )
        object_rect_to_silo(silo, object_number++, A, object);

close_silo( silo );
```

In the next example we show how objects stored in an image silo can be positioned in an image:

```
silo = open_silo( "test.silo" );  
  
int num;  
num = get_free_entry(silo);           Get number of objects in silo  
  
for( object_number=0;object_number<num;object_number++)  
    part_from_silo(silo, object_number, D, 40, 40 );
```

Finally, we show how an image silo can be presented in the form of a composite image. Note that, although not shown here, individual objects can be added to a composite image as well.

```
long comp;  
clear_im(D);  
comp = start_comp( D );  
silo_to_comp( silo, comp, 1, 100 );
```

Note that we have shown the examples as C-code fragments. In the SCIL_Image package these fragments can be entered in the interpreter literally. When using Image without SCIL, these fragments will have to be part of a complete C-program. AIO is not meant as a complete measurement system. It simply provides useful primitives which can be used in C programs. It should also be noted that more complete documentation of the individual functions can be found in the on-line manuals.

Interactive measurement

The point and click measurement of objects in AIO is (of course) only possible when a user-interface is present. In the SCIL_Image package, a routine is supplied that returns the object that the mouse is pointing at, the `point_object()` function.

The following example shows how the results of the optical density measurement can be obtained by pointing with the mouse at objects in image C. It is assumed the first part of the previous sample session (at least up to the `list_label()` command has been executed.:

```
while( object = point_object( c, o_list ) )  
    printf("sum = %f mean = %f stdev = %f\n",  
        od_sum(object), od_mean(object), od_stdev(object));
```

Implementation of the interaction part

The Image Library is designed to be independent of a user-interface, as a consequence the `point_object()` is not a default part of the AIO package. It is however a standard function of the SCIL_Image program. To allow for interactive measurement in other user-interfaces, the implementation of this function in the SCIL_Image package is supplied as an example. Please note that the `point_im()` and `MousePress()` functions are not a part of Image library either. Their functionality is discussed further down.

The `point_object()` function

The function is fairly straightforward, in a continuous loop the location of mouse-clicks are received from the `point_im()` function (#1) until a key is pressed (`c` unequal to zero). It is then checked if the mouse-click is indeed a mouse-down click in the correct image (#2). The pixel value from that location in the image is retrieved (#3) and `find_object()` translates this into the handle to the corresponding object in the object-list (#4). This handle is returned to the caller (#5) which can use it to measure the object using other AIO-functions.

```
#include "image.h"
#include "im_error.h"
#include "im_aio.h"

LIST *point_object(IMAGE *im, LIST *link)
{
    IM_EVENT      but;
    int           c, x, y;
    IMAGE         *ip;
    PIXEL         pix;
    LIST          *lp;
    LABEL_2D_IMAGE *tmp;

    im_begin_func("point_object");

    while( 1 ){
        if( c = point_im( &ip, &x, &y, &but ) ){ /* #1 */
            im_end_func("point_object");
            return( NULL );
        }
        if( im != ip || !MousePress(but) ) /* #2 */
            continue;
        tmp = (LABEL_2D_IMAGE *) ImageIn(im);
        pix = *(Label2dImageData(tmp) + /* #3 */
              (long) y * Label2dImageWidth(tmp) + x);
        if (lp = find_object(link, x, y, pix)){ /* #4 */
            im_end_func("point_object");
            return( lp ); /* #5 */
        }
    }
}
```

When called the `point_im()` function, waits until a mouse-event in an image occurred or a key on the keyboard was pressed. Its prototype:

```
int point_im(IMAGE **ip, int *x, int *y, IM_EVENT *but);
```

When it returns on a key-press of the key-board, it returns the character value of that key (the values of **ip**, **x**, **y** and **but** are not valid then). When a mouse-event happens in an image, the image, the location of the pointer in image-coordinates, and the mouse status/event are returned through the **ip**, **x**, **y** and **but** parameters.

The function `MousePress ()` tests if the event is a mouse-down event, its prototype:

```
int MousePress(IM_EVENT but);
```

It returns which of the mouse-buttons was pressed, testing for a non-zero value means that pressing any mouse-button is accepted as valid input.

Chapter 11 Bitmapped binary images

In this chapter we present methods for fast morphological binary image processing using a bitmapped representation of binary images instead of representing binary images as bitplanes (inserted in grey value images). The bitmap data structure is a very efficient representation, both in terms of memory requirements as in terms of algorithmic efficiency as the CPU operates on 32 pixels in parallel. The algorithms described in this chapter are capable of performing the basic morphological image transforms using structuring elements of arbitrary size and shape. In order to speed up morphological operations with respect to commonly used, large, convex structuring elements, the *logarithmic decomposition* of structuring elements is used.

Do not read this chapter:

- If you are a beginning user who just wants to use the binary or morphology operations.

You should read the chapter:

- If you are interested in background information on implementational aspects of mathematical morphology
- If you want to know more about bitmapped binary images.
- If you are developing binary image operations.

Introduction

Mathematical morphology as introduced by Serra [SERRA] is nowadays an important component in almost any software package for image processing. For an introduction to mathematical morphology we refer to tutorials [MARAGOS][HARALICK]. Morphological image transforms are used in many diverse application areas in image processing and computer vision. Most morphological transforms are constructed by elementary morphological operations such as erosion, dilation and hit-or-miss transform. The speedup of the elementary operations is therefore important in many applications.

As a typical example consider the program to analyze drawings of electronic schemes, called 'schema' located in the demo directory. In this program more than 50 times an elementary morphological operation is used. Other examples show the same pattern.

In Image we provide a new algorithmic implementation of the basic binary morphological operations on general purpose sequential computers which are between 10 to 50 faster than existing fast implementations ([VERWER], [GROEN]).

In contrast to the previous fast implementations, the provided algorithms do admit the use of structuring elements of arbitrary shape (within a 65x65 neighborhood). This is based on our belief that implementations of the basic morphological transforms should be valid for arbitrary structuring elements without strong performance degradation, as non-circular structuring elements prove to be very efficient in solving specific problems.

The speed of common implementations of morphological image transform in most cases, if not all, is linear proportional to the number of pixels in the structuring element. We use logarithmic decomposition of structuring elements to efficiently process large, convex structuring elements. Compared with the standard linear decomposition logarithmic decomposition results in much faster algorithms.

Erosions, Dilations and Logarithmic Decomposition

The basic image operations defined in mathematical morphology are the erosion and dilation. Let $A[k,l]$ be a binary image (for $0 \leq k \leq K$, $0 \leq l \leq L$). By definition we say that a pixel (k,l) is an object pixel if $A[k,l]=1$. If $A[k,l]=0$ the pixel (k,l) belongs to the background. A structuring element is also a binary image (most often a small one). The erosion of the image A with structuring element S resulting in the image B is the neighborhood operation defined as:

$$B[k,l] = \text{AND}_{S[i,j]=1} A[k-i,l-j] \tag{1}$$

i.e. we take the logical AND of all the pixels in the neighborhood of (k,l) as defined by the structuring element S. The above definition states that a pixel (k,l) is an object pixel in the erosion result only if all pixels in the neighborhood defined by the structuring element S centered at (k,l) are object pixels in the original image A.

The dilation of the image A with structuring element S is defined as:

$$B[k,l] = \text{OR}_{S[i,j]=1} A[k-i,l-j] \tag{2}$$

i.e. instead of AND-ing all pixels, as is done in the erosion, we take the logical OR of the pixel values.

The computational complexity of the erosion and dilation is linear proportional to the number of pixels in the structuring element. For a square structuring element of NxN pixels the time complexity thus is quadratic in N. For large N the decrease in speed therefore is dramatic. Simple but often used structuring elements however can be decomposed into smaller ones, and so the time complexity can be reduced. In this section we will only use the 4-connected neighborhood. This structuring element will be denoted as D.

To illustrate 3 different approaches to decompose convex structuring elements, consider the decompositions of 8D. The well-known and often used, linear decomposition gives (the total number of pixels in the structuring elements is 40):

$$8D = \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\} \oplus \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\} \oplus \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\} \oplus \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\} \oplus \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\} \oplus \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\} \oplus \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\} \oplus \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\}$$

Thus instead of eroding/dilating with a large (17x17) structuring element we can erode/dilate 8 times using a 3x3 structuring element. The linear decomposition using the extreme set E(D) gives (total number of pixels considered is 33):

$$8D = \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\} \oplus \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\} \oplus \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\} \oplus \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\} \oplus \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\} \oplus \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\} \oplus \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\} \oplus \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\}$$

The logarithmic decomposition finally gives (total number of pixels considered is 17):

$$8D = \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\} \oplus \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\} \oplus \left\{ \begin{matrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \right\} \oplus \left\{ \begin{matrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{matrix} \right\}$$

It can be proven that logarithmic decomposition can be used for all convex structuring elements. For the 4-connected and 8-connected neighborhoods, logarithmic decomposition is automatically taken care of in Image.

Algorithmic Implementation

Data Representation

In most modern general purpose computers the 'Von Neuman' bottle-neck still exists. The efficiency of low level image processing operations is bound by the speed at which the image data can be transported between the memory and the CPU.

Traditionally in image processing binary images are stored as one bitplane in a NxM pixel mapped grey value image. In a pixel mapped image the grey value of each pixel is stored in one computer addressable memory word. A binary image, needs only one bit per pixel to indicate whether that pixel is part of the object or whether it is a background pixel. To access the value of one bit in a bitplane the CPU has to fetch all the bits of one pixel (typically 8 or 16). All irrelevant bits (7 out of 8, or 15 out of 16) have to be masked out. Using a 32 bit computer just to fetch 1 relevant bit at a time is a 'bit' overdone. A second disadvantage is that even if only 1 binary image is needed still the memory for 8 or 16 bitplanes has to be allocated.

In Image a bit mapped representation of binary images is used. Each 32 bit addressable word contains the bit values of 32 horizontally consecutive pixels. A 32-bit computer is optimally used in the sense that all 32 bits fetched in one memory cycle are relevant. Not only 32 pixels in parallel are transported from and to memory, but also the operations are done on 32 pixels in parallel. Furthermore there is no need to allocate more memory than is actually needed to represent the binary image. Figure 11-1 visualizes the differences between a pixel mapped and a bit mapped binary image.

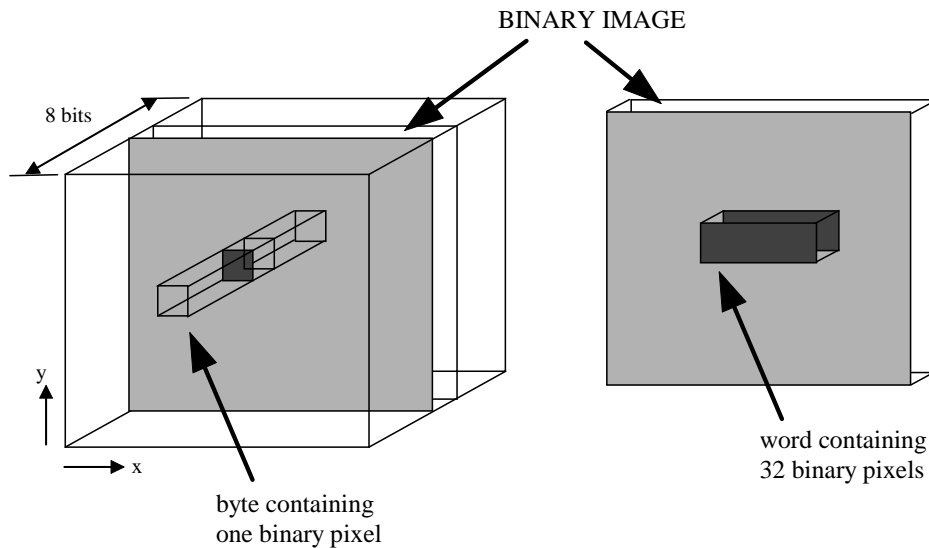


Figure 11-1 Pixelmapped versus Bitmapped Images

All the 32-pixel words are stored in a linear array in the computer memory. To facilitate the development of efficient algorithms we have chosen not only to store the pixels in the image itself, but also a border around the actual image.

Figure 11-2 shows the linear array (depicted in a two-dimensional layout). The numbers in the words are the indices in the linear array. The border words depicted at the right of the image also serve as 32-pixel border at the *left* of the actual image (word 17 is the right neighbor of word 16 and the left neighbor of word 18). The word with index -1 is needed to store the north-west neighboring pixel of the left most pixel in word 9.

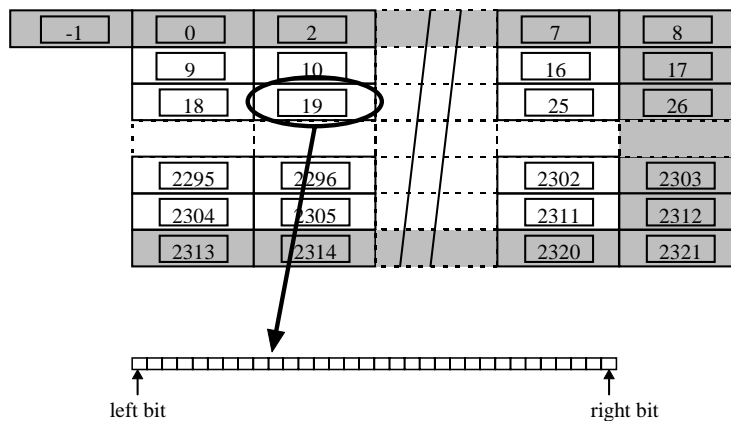


Figure 11-2 Bitmapped Image Representation

Only a small part of a binary image of 256x256 pixels is shown. The shaded words are not part of the actual image but define a border around the image. The word with index 19 is magnified to show the 32 pixels within one word

Any image representation has to deal with the fact that structuring elements positioned at the edge only partly overlap with the image. In the algorithms, pixels not within the actual image are set to either 1 or 0 depending on the operation.

Note however that the border around the actual image which is also stored in memory makes "border checking" *not* necessary for structuring element pixels with coordinates (k,l) satisfying: $|k| \leq 32$ and $|l| \leq 1$. In the algorithms described in this chapter we assume that $|k| \leq 32$ necessitating border checks only for those pixels in a structuring element for which $|l| > 1$. This choice is no essential restriction due to the chosen border and can be easily lifted at the expense of extra border checks.

Although structuring elements are sets and could have been represented with bitmaps, we have chosen another representation which can be more efficiently dealt with in the algorithms described in the next subsection. Figure 11-3 depicts the data structure used to represent arbitrary structuring elements. The "indicator" array F(i) is used to represent a hit-or-miss mask (S,T) in one structure. By definition a pixel with index i is element of S if $F(i) > 0$ and element of T if $F(i) < 0$. The i^{th} element of a hit-or-miss mask is described with two coordinates (X(i),Y(i)) and its indicator F(i).

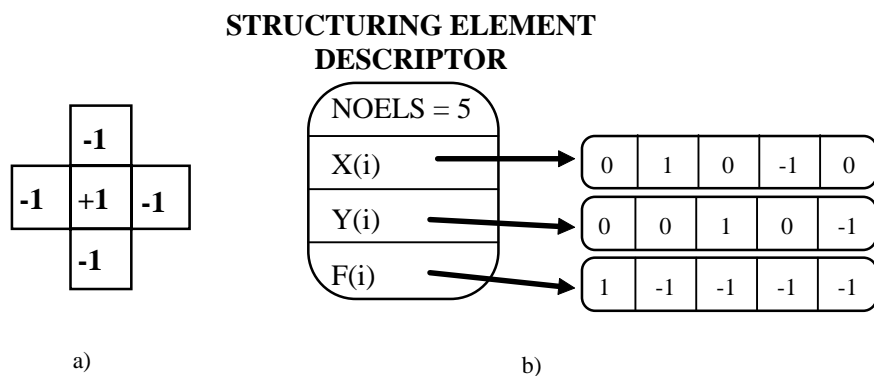


Figure 11-3 Hit-or-miss Mask Representation
 In a. an example of a simple hit-or-mask (S,T) is shown with pixels in S denoted as "+1" and those in T as "-1". In b. the corresponding mask representation is shown.

Implementation of the Pixelwise Logical Operations

The pixelwise logical operations of one or two binary input images are the Boolean equivalents of the union, intersection and complementation of sets. Using bitmapped binary images these operations can be performed very fast, as 32 pixels are operated upon in parallel.

Algorithm 1 shows the pseudo code to perform the logical AND of two binary images (we use the C-syntax to denote AND with '&'). Replacing '&' with '|' (denoting the bitwise OR) in algorithm 1 gives the set union.

Note that not only the image itself is processed, but also the border. This small amount of extra work makes nested for-loops superfluous.

ALGORITHM 1 Logical Operations
 Shown is the pseudo-code to perform the logical AND of two images (represented in bitmaps B and C) resulting in bitmap D.

```
FOR i=0 TO MaxIndex DO
    D[i] = B[i] & C[i]
ENDDO
```

Implementation of the Morphological Operations

Rather than operating on one central pixel in combination with neighboring pixels, the bitmap organization of binary images allows us to operate on 32 central and 32 neighbor pixels in parallel. This is simple in case we operate on the central pixel and the pixel above, or below because if $A(i)$ is the word containing 32 central pixels, $A(i-\text{LINEOFFSET})$ and $A(i+\text{LINEOFFSET})$ contain the 32 pixels just above, respectively just below the central pixels. However if we operate on a central pixel with the neighbors to the left, or to the right it gets more complicated.

Again assume that $A(i)$ contains the 32 central pixels then the pixels just to the left of these central pixels are not in one word in the bitmap. The left neighbor of the leftmost pixel in $A(i)$ is in the word $A(i-1)$. In order to represent all the 32 left neighboring pixels we introduce the so called BARRELSHIFT function.

The function $\text{BARRELSHIFT}(A, i, k, l, \text{border})$ is given in pseudo code in algorithm 2. We use the C-notation (\ll , \gg) to denote shifting of the bits in an integer.

ALGORITHM 2 **Barrelshift Function**

Pseudo code for the Barrelshift function, where A is the bitmap, i is the bitmap index of the central pixels, (k, l) is the pixel position relative to these central pixels and **border** is the word returned when pixels outside the image are accessed.

```
BARRELSHIFT( A, i, k, l, border )
DO
    j = i + l * LineOffset
    IF ( 0 < j < MaxIndex )
        // The pixels are within the bitmap
        IF ( k = 0 )
            // No Barrelshifting needed
            RETURN ( A[j] )
        ELSE IF ( k < 0 )
            // we are looking for pixels to the left
            // of the central pixel
            RETURN ( ( A[j] >> abs(k) ) | ( A[j-1] << (32+k) ) )
        ELSE
            // we are looking for pixels to the right
            // of the central pixel
            RETURN ( ( A[j] << k ) | ( A[j+1] >> (32-k) ) )
    ELSE
        // the pixels are outside the bitmap
```

```
        RETURN ( border )
    ENDDO
```

The pseudo code for the erosion using an arbitrary structuring element is given in algorithm 3. In the algorithms the border effects (when the structuring element is not entirely within the image) are not accounted for. Replacing the logical AND (&) in algorithm 3 with a logical OR (!) results in an algorithm for the dilation.

ALGORITHM 3 Erosion
The pseudo code for the erosion of an image in bitmap A resulting in an image in bitmap B. The structuring element is encoded in the arrays X,Y, and F.

```
EROSION( A, B, X, Y, F )
DO
    i = LineOffset // index the first word in the actual image
    FOR y=0 TO L-1 DO // L is the size of the image in y-direction
        FOR x=0 TO n/32 DO
            result = BITWORDALL // all bits set to 1
            FOR j=0 TO Noels-1 DO
                IF F[j]>0
                    result = result & BARRELSHIFT( A,i,X[j],Y[j],1)
            ENDDO
        ENDDO
        B[i] = result
        i = i+1 // index the next word
    ENDDO
    i = i+1 // skip over the words in the border
ENDDO
```

Evaluation

In this section the performance of the proposed algorithms will be compared with the performance of existing implementations. The new algorithms will be referred to as the "BITMAP"-algorithms. They will be compared with two software implementations and with a Special purpose Image Processing machine (KONTRON) which will be referred to as "SIP".

All our BITMAP-algorithms are coded in portable C. Tests were performed using a SUN SPARC station 1, except for the results obtained with the SIP. All the timing results are obtained by averaging the results over at least 20 experiments.

Because some implementations of the morphological transforms are data dependent, we have chosen a set of three test images. These are depicted in Figure 11-4.

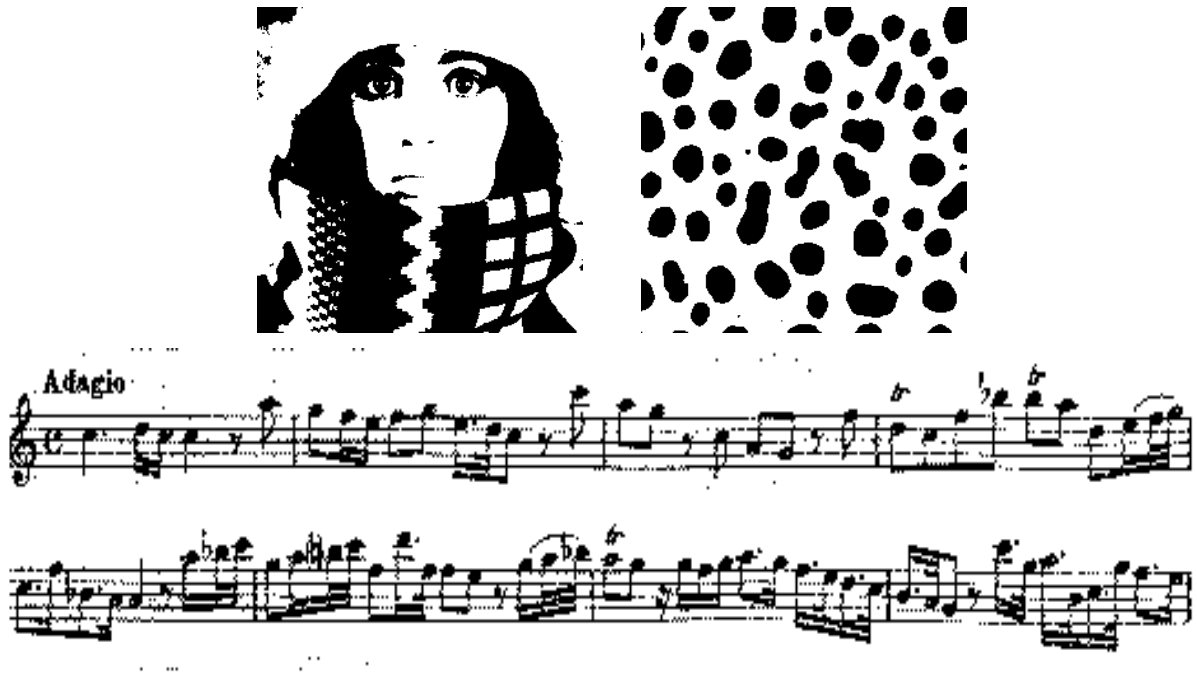


Figure 11-4. Test Images
 a) Trui (256x256) (top left)
 b) Cermet (256x256) (top right)
 c) Musicscore (1024x244) (bottom)

Pixelwise Logical Operations

The pixelwise logical operations on bitmapped images are much faster than the equivalents working on pixel mapped images. This is due to the fact that instead of operating on one pixel value at the time, we operate on 32 pixels in parallel. There is also no need to isolate the pixel bit from the grey pixel value fetched from memory.

Table 11-1 shows the experimental results for a 256x256 binary image. The algorithm for the pixel mapped images is denoted as "PIXELMAP". The table also shows the results for the SIP machine.

Operation	<i>BITMAP</i> <i>(ms)</i>	<i>PIXELMAP</i> <i>(ms)</i>	<i>SIP</i> <i>(ms)</i>
AND	2.0	87.3	93.0
OR	1.7	87.3	93.0
EXOR	1.7	115.0	91.0
INVERT	1.3	74.0	N.A.
COPY	1.3	74.3	46.0

Table 11-1 Pixelwise Logical Operations
 Timing results for the AND, OR, EXOR, INVERT and COPY operations

Morphological Transforms

The bitmapped implementations of the erosion are compared with two existing software implementations and with the special purpose image processing machine (SIP). The most simple method to implement erosions and dilations using structuring elements of size 3x3 is to use look-up tables. A table with 512 entries holds the results (for erosion, dilation, etc.) for all possible configurations in a 3x3 neighborhood. Transforming the image thus boils down to one table-look action per image pixel. This method will be denoted as the Straight Forward Implementation (SFI).

We will also compare our algorithms with the queue based algorithms developed by Verwer and VanVliet ([VERWER]). These algorithms will be referred to as the VVV-algorithms. These algorithms process only those pixels in the image which need to be processed. The first iteration of erosions and dilations using the VVV algorithm is slow because the queue containing the pixels to be processed needs to be build by scanning the entire image. Subsequent iterations then are much faster (proportional with the number of contour pixels).

Number of Iterations	<i>BITMAP</i> (ms)		<i>VVV</i> (ms)			<i>SFI</i> (ms)		<i>SIP</i> (ms)	
	a,b	c	a	b	c	a	b	a,b	c
1	6	23	175	160	639	283	283	153	1216
2	11	43	187	171	723	583	516	273	2416
4	15	62	206	200	851	1150	1133	491	4900
8	19	82	237	255	945	2300	2366	858	9850
16	24	100	269	302	1089	4716	4800	1371	20066
32	28	121	288	305	1303	9600	7000	2155	41616
64	36	153	291	305	1414	9600	6983	3643	86016
128	44	187	291	305	1423	9750	7050	7178	151066

Table 11-2: 4-Connected Dilation

Shown are the execution times (in ms) for the 4-connected dilation for the 3 different original images (denoted as a,b and c, see Figure 11-4).

Table 11-2 shows the experimental results for the 4-connected dilation for several values of n, where n is the number of iterations For the BITMAP algorithms only the results using the logarithmic decomposition are shown.

Discussion and Conclusions

In this chapter we presented new algorithms for binary morphological transforms based on a bitmapped representation of binary images and the use of logarithmic decomposition of

structuring elements. The new algorithms prove to be on average one order of magnitude faster than existing fast implementations.

The results presented in this chapter indicate that bitmapped representation of binary images is well-suited for morphological image processing, as it is very efficient both in terms of memory requirements and in terms of algorithmic efficiency. Instead of the necessity to allocate at least 8 binary images (when using the pixel mapped representation) only the bits in one binary image are stored. When working with very large images like those acquired with page scanners, the reduced memory requirement is an important feature.

In order to deal with large structuring elements (obtained from auto-dilation of smaller structuring elements) logarithmic decomposition proves to be a powerful tool.

Literature

- [HARALICK] R.M. Haralick, S.R. Sternberg, X. Zhuang, "Image Analysis using Mathematical Morphology", IEEE Transactions on Pattern Analysis and Machine Intelligence, **PAMI-9**, 532-550, July 1987
- [MARAGOS] P. Maragos, "Tutorial on Advances in Morphological Image Processing and Analysis", Optical Engineering, **26**, 7, 623-632, July 1987
- [VERWER] B.J.H. Verwer, L.J. VanVliet, "A Contour Processing Method for Fast Binary Neighborhood Operations", Pattern Recognition Letters, 7, 27-36, 1988
- [GROEN] F.C.A. Groen, N.J. Foster, "A Fast Algorithm for Cellular Logic Operations on Sequential Machines", Pattern Recognition Letters, 1988
- [SERRA] J. Serra, "Image Analysis and Mathematical Morphology", Academic Press, 1982

Chapter 12 New image types

This chapter describes how a user can create subtypes of existing image types and how new image types can be implemented in Image as if they were standard types.

Read this chapter :

- If you want to implement a new image type.
- If you want to know more about the image infrastructure.

Do not read this chapter :

- If you are not an experienced C-programmer.
- If you have not read "**Programming with Image**".

Introduction

The infrastructure of Image supports a strong separation of the image types for reasons of type safety and extensibility. To enable the definition of new image types without having to change the infrastructure, requires that no knowledge of image type details are incorporated in the infrastructure. This means that all image type specific actions must be performed by functions tailored for the image type. Calling the correct function for each image type is accomplished by means of function overloading.

When adding new image types to Image we distinguish two situations:

- 1) **A new image type.** The image type has a data representation different than that of existing image types. In this case the steps outlined in "Implementing a new image type" must be taken.
- 2) **A subtype of an existing image type.** The image type has the same data representation as an already implemented image type but the data is to be interpreted according to the application specific semantics. Therefore it will have a specific set of operations associated with it. In this case the new image type can make use of functions of the already implemented image types thereby reducing the number of new functions to be written. The image types that make use of functions from another image type are referred to as subtypes.

Implementing a new image type

Implementing a new image type in Image means that you must supply a number of required *service* functions for the new image type, namely:

create_image	allocate the data space and fill in the type specific data structures.
destroy_image	free the data space and associated structures.
copy_part_image	copy a rectangular sub-image to another image.
copy_masked_part	copy a part masked by a Boolean mask to another image.
convert_to_common	put the image data in a COMMON_LINE.
convert_from_common	read image data from the COMMON_LINE.
pix_value_str	put single pixel data in a string
get_image_window_info	put info on image in a string for use in a title bar
part_image_display	display a rectangular part of the image

In this section we will explain the process of adding a new image type by implementing the byte 2d image type, a two dimensional image with 8 bit pixels.

The basic functions for the standard image types: GREY_2D, BINARY_2D and FLOAT_2D are made available in the "src_exmp" directory in the standard distribution . These functions serve as a starting point to develop type specific service functions. In the 'basic.c' and 'basiclow.c' files in these directories the interface functions for these image types are gathered. It is important to understand that the interface functions are accessed through the overloading mechanism. Therefore, sometimes some extra parameters are in the function argument list.

Please note that the last three functions are only required when implementing a new image type in the complete SCIL_Image environment. These three functions are needed by the display-interface of SCIL_Image, the Image library does not need them to support the image types in stand-alone applications.

Defines and structures

In order for the infrastructure to recognize the new image type, a `type_ident` and a `type_spec` must be defined. The `type_spec` is a number that is unique for the data representation of the image (it must be an unique bit in a 32 bit word compared with the other image types). The `type_ident` must be a unique number that identifies the image type. In the header file 'image.h' the `type_specs` and `type_idents` of the standard image types are listed. For the byte 2d image type we choose a `type_spec` of 1024 (10th bit set) and a `type_ident` of 11. Both of these values are currently unused in the standard set of image types

A structure with image type specific information must be created. This structure will be linked to the `in_descript` and `out_descript` pointers of the `IMAGE` structure. The first two fields of this structure **must** be an `int` that contains the `type_ident` of the image type and a pointer to the image data (`BYTE *`). The dimensions of the image must also be stored in this structure. The byte image type does not need any additional information so the structure is complete. It is advisable to define macros to access the individual fields of the structure which can be used in the source code.

The defines, the structure and the macros are best put in a file which is named after the image type, 'BYTE' is already defined in 'image.h'. In this example we store them in the file 'byte_2d.h' :

```
#ifndef BYTE_2D_H      /* necessary to avoid multiple definitions/  
#define BYTE_2D_H    /* file inclusion */  
  
#define BYTE_2D      11    /* unique value */  
#define BYTE_2D_SPEC 1024L /* unique bit field */
```

```
/* image type specific type descriptor */
typedef struct byte_2d_t {
    int     type;          /* mandatory field! */
    BYTE   *data;        /* mandatory field! */
    int     lenx;
    int     leny;
} BYTE_2D_IMAGE;

/* Macro's to allow for access */
#define Byte2dImageType(bp)      (bp)->type
#define Byte2dImageData(bp)     (bp)->data
#define Byte2dImageWidth(bp)    (bp)->lenx
#define Byte2dImageHeight(bp)   (bp)->leny

#endif /* BYTE_2D_H */
```

The prototypes of all the function for the image-type are kept in a separate include file "byte_2dp.h". Its contents is listed in the "Low Levels" paragraph on page 12-15.

Creation and destruction

The first two functions needed for a new image type are the create and destroy functions. The infrastructure uses these functions to create this type of image and to dynamically adjust the type of an existing image. Every image processing operation specifies the type and sizes of the output image. These functions are called by the infrastructure to adjust the image to the operation's demands as required.

The task of the create-image function is to allocate a type descriptor (the type description structure), the space needed for the image and to fill in the structure. A pointer to the newly created type descriptor must be returned on success. When either of the two allocations fail, the already allocated space (if any) must be freed and a NULL pointer must be returned.

Function overloading requires the function header to have the following form:

```
type_structure *type_create(char *name, int type,
                             int lenx, int leny, int lenz);
```

The create-image function for the byte 2d image type is defined as follows :

```
#include <stdlib.h>
#include <stdio.h>
#include "image.h"
#include "im_error.h"
#include "im_infra.h"
#include "imtypinf.h"
#include "roi.h"
#include "byte_2d.h"
#include "byte_2dp.h"

BYTE_2D_IMAGE *byte_2d_create_image(char *name, int type,
                                   int lenx, int leny, int lenz)
{
    BYTE_2D_IMAGE *bp;

    im_begin_func("byte_2d_create_image");
```

```

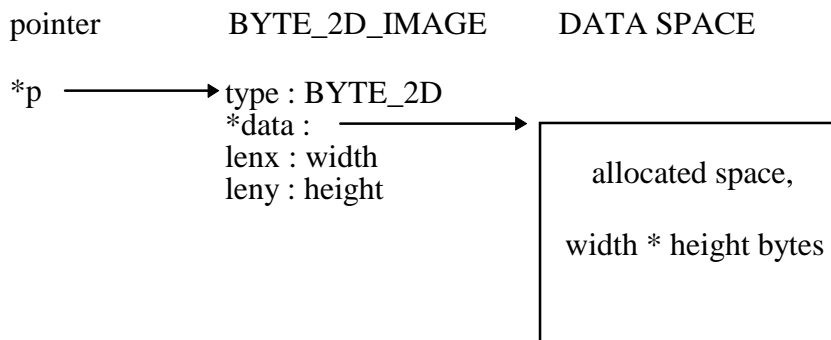
if ((bp = New(BYTE_2D_IMAGE)) == NULL) {
    im_report_error("byte_2d_create_image", IE_NOMEM,
        "couldn't allocate BYTE_2D image");
    return NULL;
}
if (lenz != 1) {
    image_output(IMO_OUTPUT,
        "byte_2d_create_image: lenz %d overruled, set at 1\n", lenz);
    lenz = 1;
}
Byte2dImageType(bp) = type;
Byte2dImageData(bp) = (BYTE *) calloc( lenx * (long) leny * lenz,
    sizeof(BYTE));

if (! Byte2dImageData(bp)) {
    free(bp);
    im_report_error("byte_2d_create_image", IE_NOMEM,
        "byte_2d_create_image unable to allocate image data");
    return (NULL);
}
Byte2dImageWidth(bp) = lenx;
Byte2dImageHeight(bp) = leny;

im_end_func("byte_2d_create_image");
return (bp);
}

```

The diagram below shows what the function should create.



The destroy-image function is responsible for freeing the space allocated by the create-image function. In general it will free the data space occupied by the image and then the type description structure. The function has one argument, a pointer to a type descriptor, and no return value. Its mandatory function header is :

```
void type_destroy(type_descriptor pointer)
```

The byte_2d_destroy_image function: .

```

void byte_2d_destroy_image(BYTE_2D_IMAGE *bp)
{
    im_begin_func("byte_2d_destroy_image");

    if (bp) {
        if (Byte2dImageData(bp))
            free(Byte2dImageData(bp));
        free(bp);
    }

    im_end_func("byte_2d_destroy_image");
    return;
}

```

Copying a part of the image

Regions Of Interest (ROI) are handled entirely by the infrastructure. An image processing operation does not have to perform any special action to process a region of interest as long as it calls `pre_op()` before it starts to process the data. The infrastructure (`pre_op()`) copies the data from the parent of the ROI into a separate image. The ROI image can then be processed by the operation. Finally, if the output image is also a ROI, `post_op()` copies the data from that ROI image back into its parent.

For copying the image data from the parent to the ROI and back to the parent the functions `copy_part_image()` and `copy_masked_part()` are used. The first one for rectangular ROIs and the latter for arbitrarily shaped ROIs. The infrastructure uses the create and destroy functions to allocate the necessary data space and then the copy functions to put the data in.

The `copy_part_image()` function is supplied with the position of the rectangle, the sizes and the destination position which are all already checked for validity on a higher level. As the function is also called from other functions, it must include calls to `pre_op()` to check and adjust the images it receives. The input image must be COMPARED with itself and the output image must be ADJUSTed to match the input image. As the output image was ADJUSTed with `pre_op()`, it must also be given to `post_op()` The function must return OK (1) if it was successful and NOT_OK (0) otherwise. Its function heading and the calls to `pre_op()` and `post_op()` are :

```
int type_copy_part_image(IMAGE *in, IMAGE *out,
                        int sx, int sy, int sz;
                        int width, int height, int depth;
                        int dx, int dy, int dz;
{
    if(!pre_op(in, in, COMPARE, type_spec, ImageTypeId(in)) ||
        !pre_op(out, out, ADJUST, ImageTypeSpec(out),
                ImageTypeId(in)) )
        return( NOT_OK );

    /* the actual copying */

    return( post_op( out ) );
}
```

For the byte 2d image type the implementation of the routine uses two lower level routines (`l_byte_read_part_image()` and `l_byte_write_part_image()` see "Low Levels" on page 12-15) that copy the image data to and from an newly allocated piece of memory. Although this is not obligatory, these lower level routines can be useful for other routines that also might want to copy a part of an image (these low-levels are listed at the end of this section).

```
int byte_2d_copy_part_image(IMAGE *in, IMAGE *out,
                            int sx, int sy, int sz,
```



```

        int width, int height, int depth,
        int dx, int dy, int dz)
{
    BYTE *ptr;
    int status;

    im_begin_func("byte_2d_copy_part_image");

    if(!pre_op(in, in, COMPARE, BYTE_2D_SPEC ,ImageTypeId(in) ) ||
        !pre_op(out, out, ADJUST, ImageTypeSpec(out),ImageTypeId(in)) )
        return( im_report_error("byte_2d_copy_part_image",
                                IE_PRE_OP,"") );

    if ( !(ptr = l_byte_read_part_image(ImageInData(in), sx, sy, sz,
        width, height, 1, ImageWidth(in), ImageHeight(in),
        ImageDepth(in)) ) ) {
        return(im_report_error("byte_2d_copy_part_image", IE_NOMEM,
            "Unable to allocate part image buffer"));
    }

    l_byte_write_part_image(ImageOutData(out), ptr, dx, dy, dz,
        width, height, 1, ImageWidth(out),
        ImageHeight(out), ImageDepth(out));
    free(ptr);

    if((status = post_op(out)) != IE_OK)
        return(im_report_error("byte_2d_copy_part_image", status,""));

    im_end_func("byte_2d_copy_part_image");
    return ( IE_OK );
}

```

The `copy_masked_part()` is responsible for copying arbitrarily shaped Regions Of Interest from and to the parent. To accomplish this, the function is supplied with two more parameters than the `copy_part_image()` function.

The second parameter of the function is a Boolean mask. The bits which are set in the mask indicate which pixels from the rectangle of the ROI in the parent must be copied to the ROI image. An extra flag 'clear' is added to indicate what to do with the pixels in the rectangle of the ROI that are not in the Boolean mask. This flag is used by `pre_op()` and `post_op()` when copying the ROI data from the parent to the ROI image and back to the parent. `pre_op()` calls the function with 'clear' set (unequal to zero) to ensure that the pixels in the **ROI image** that are not covered by the Boolean mask are cleared. `post_op()` calls the function with 'clear' equal to zero to ensure that the pixels in the **parent** that are not covered by the Boolean mask keep their value.

The parameter list and the calls to `pre_op()` must be :

```

    int type_copy_masked_part(IMAGE *in,BOOL_MASK *mask,IMAGE *out,
        int sx, int sy, int sz,
        int width, int height, int depth,
        int dx, int dy, int dz, int clear)
    {
        if(!pre_op(in, in, COMPARE, type_spec, ImageTypeId(in)) ||
            !pre_op(out, out, ADJUST, ImageTypeSpec(out),
                ImageTypeId(in)) )
            return( NOT_OK );
    }

```

```
        /* the actual copying */
        return( post_op( out ));
    }
```

The `byte_2d_copy_masked_part` function:

```
int byte_2d_copy_masked_part(IMAGE *in, BOOL_MASK *mask, IMAGE *out,
int sx, int sy, int sz, int width, int height, int depth,
int dx, int dy, int dz, int clear)
{
    BYTE *ptr;
    int status;

    im_begin_func("byte_2d_copy_masked_part");

    if(!pre_op(in,in,COMPARE,BYTE_2D_SPEC,ImageTypeId(in) ) ||
        !pre_op(out, OUT, ADJUST, ImageTypeSpec(out),ImageTypeId(in)) )
        return(im_report_error("byte_2d_copy_masked_part", IE_PRE_OP,""));

    if ( !(ptr = l_byte_read_masked_part(ImageInData(in), mask,
        sx, sy, sz, width, height, 1, ImageWidth(in),
        ImageHeight(in), ImageDepth(in)) ) ) {
        return(im_report_error("byte_2d_copy_masked_part", IE_NOMEM,
            "Unable to allocate part image buffer"));
    }

    l_byte_write_masked_part( ImageOutData(out),ptr, mask, dx, dy, dz,
        width,height,1,ImageWidth(out),
        ImageHeight(out),ImageDepth(in),clear);

    free(ptr);

    if ((status = post_op(out))<IE_OK)
        return(im_report_error("byte_2d_copy_masked_part", status,""));

    im_end_func("byte_2d_copy_masked_part");
    return(IE_OK) ;
}
```

Displaying the image

To view the image on the monitor in the `SCIL_Image` package, a function must be implemented to display the image. The function must meet the following requirements:

- It **must** be able to display an arbitrary rectangular part of the image (needed for ROI display). The parameters of the function specify the image, the origin and the sizes of this rectangle.
- It **must** write the data to a swap area of the `IMAGE` structure. This swap area (used for handling expose events) is allocated by the infrastructure when `make_image` is called, a pointer to it is located in the `viewport` structure that is attached to the `IMAGE` structure.

- It **must** be able to handle any window size both the x and y dimensions may differ from the image dimensions. When the window is resized, the infrastructure will free the swap area and allocate a new one to match the size of the window.
- It **must** call the function `PwWrite()` to actually display the rectangle written into the swap space.

The display routines of the standard image types in `SCIL_Image` handle :

- Intensity scaling, which can be set to different modes as specified in the `set_display_mode()` function. The scaling mode of an image is stored in the `mode_flags` field of the `VIEWPORT` structure. Its value is defined in the `'dmodes.h'` header file.
- Various monitors, namely 1 bit monochrome, 8 bit color and 24 bit full color monitors (**Macintosh**: only 8 bit color is supported).
- Color lookup tables (CLUT) which can be attached to an image. The `set_dither_mode()` function can be used to improve the quality of the display of colors on a 8 bit monitor. The dither-mode flag is also a member of the `mode_flags` field of the `VIEWPORT` structure.

The function header of the display function must be :

```
int type_part_image_display(im, sx,sy,sz, width, height, depth)
IMAGE *im;
int    sx, sy, sz, width, height, depth;
```

The byte 2d example, for sake of simplicity, is only capable of size-scaling and color lookup tables and does not consider the automatic display enhancements attached to the image. For a complete description one should refer to the display sources as provided with the standard distribution. It is also assumed that the standard configuration is an 8bit screen, for 1bit and 24bit monitors again the reader is referred to the source examples.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "image.h"
#include "im_error.h"
#include "im_infra.h"
#include "portab.h"
#include "dmodes.h"
#include "disp_im.h"
#include "byte_2d.h"

#ifdef MACINTOSH
#include <Palettes.h>
#include "Macviewport.h"
#include "disp_mac.h"
#else
#include "xviewprt.h"
#include "disp_x.h"
#endif
```

```

extern CLUT          **standard_cluts;

int byte_2d_part_image_display(im, sx, sy, sz, width, height, depth)
IMAGE *im;
int    sx, sy, sz, width, height, depth;
{
    im_begin_func("byte_2d_part_image_display");

    if ( !ImageVport(im) ) {
        im_end_func("byte_2d_part_image_display");
        return(IE_OK);
    }

    if (!ImageDispClut(im) || !is_clut(ImageDispClut(im))) /* if no clut */
        ImageDispClut(im) = standard_cluts[GREY_LUT_T];

    l_byte_2d_disp_8bit(im, ImageVport(im), sx, sy, width, height);
    im_begin_func("byte_2d_part_image_display");
    return (OK);
}

/* 8 bit display routine, only scaling, no dithering */
int l_byte_2d_disp_8bit(im, vport, sx, sy, width, height)
IMAGE *im;
VIEWPORT *vport;
int    sx, sy, width, height;
{
    short          dispw, disph;
    unsigned char  *swap;
    int            part_sx, part_sy, part_width, part_height;
    short          xsize, ysize;
    short          *xtab, *ytab;
    unsigned long  dtab[256];
    BYTE_2D_IMAGE *ip;
    register BYTE  *pix;
    register short *xt;
    register BYTE  *vdi;
    register int   j;
    register int   i;
    register BYTE  tmp;

    im_begin_func("l_byte_2d_disp_8bit");
    /* sizes of the viewport */
    dispw = ViewportWidth(vport);
    disph = ViewportHeight(vport);

    /* get the pointer to the swap area */
    swap = (unsigned char *) ViewportSwap(vport);

    /* the sizes of the image */
    ip = (BYTE_2D_IMAGE *)ImageOut(im);
    xsize = Byte2dImageWidth(ip);
    ysize = Byte2dImageHeight(ip);

    /* copy the display values from the color lookup table */
    for(i=0; i<256; i++)
        dtab[i] = ViewportClut(vport)->table[i];

    /* allocate space for the size stretch tables */
    xtab = (short *) calloc(dispw, sizeof(short));
    ytab = (short *) calloc(disph, sizeof(short));
    if (!(xtab && ytab))
        fatal_err("l_byte_2d_disp_8bit");

    /* fill the size stretch tables */

```

```
skip_tab(xtab, xsize, dispw);
skip_tab(ytab, ysize, disph);

/* get the destination position of the pixels to be displayed */
part_sx = ((double)sx/xsize) * dispw;
part_sy = ((double)sy/ysize) * disph;
part_width = ((double)(width)/xsize) * dispw + 0.5;
part_height = ((double)(height)/ysize) * disph + 0.5;

/* point to the first destination position */
swap = swap + part_sy * dispw + part_sx;

for (i = 0; i < part_height; i++) {
    pix = (BYTE *)Byte2dImageData(ip) + (xsize * ytab[part_sy+i]);
    vdi = swap;
    swap += dispw;
    xt = &xtab[part_sx];
    j = part_width;
    while(--j >= 0) {
        tmp = *(pix + (*xt++));
        *vdi++ = (BYTE)dtab[tmp];
    }
}

free(xtab);
free(ytab);
PwWrite(vport, part_sx, part_sy, part_width, part_height);
im_end_func("l_byte_2d_disp_8bit");
return( IE_OK );
}
```

Image type information

In the SCIL_Image environment, the information supplied to the user in the title bar of an image window and the information in the floating window when the left mouse button is pressed in an image (auto_point information), is supplied by two small functions. In the title bar, the name of the image, its type and the sizes of the image are generally displayed. The auto point information supplies the user with coordinates and the value of the pixels he points to with the mouse.

The auto point information is stored and returned in a string format. The global string `pix_val_buf` is allocated by the infrastructure to store the string. The pointer to `pix_val_buf` should be the return value of the function if all is OK. Otherwise NULL should be returned. The function header must be :

```
char *type_pix_value_str( IMAGE *image, int x , int y, int z )
```

The byte 2d auto point information is supplied by this function :

```
char *byte_2d_pix_value_str(IMAGE *ip, int x, int y, int z )
{
    BYTE_2D_IMAGE    *bp;

    im_begin_func("byte_2d_pix_value_str");

    if (!range_ok(x, 0, ImageWidth(ip), "X coordinate") ||
        !range_ok(y, 0, ImageHeight(ip), "Y coordinate"))
```

```
        im_report_error("byte_2d_pix_value_str",IE_OUTRA, "");
        return (NULL);

    bp = (BYTE_2D_IMAGE *) ImageIn(ip);
    sprintf(pix_val_buf, "%3d", *(Byte2dImageData(bp) +
        (long) y * Byte2dImageWidth(bp) + x));

    im_end_func("byte_2d_pix_value_str");
    return( pix_val_buf );
}
```

The information to be displayed in the title bar of the window is supplied by a function that prints the type of the image and its sizes in an supplied string. The name of the image is inserted by the infrastructure itself. The function header:

```
void type_get_image_window_info( IMAGE *image, char *buf )
```

The byte 2d version :

```
void byte_2d_get_image_window_info( IMAGE *im, char *buf)
{
    im_begin_func("g_2d_get_image_window_info");

    sprintf(buf, "(byte2D) %d*%d ", ImageWidth(im),ImageHeight(im));

    im_end_func("g_2d_get_image_window_info");
    return;
}
```

Conversion to other image types

By implementing two functions per image type, one to put the data in an intermediate data space and one to retrieve the data from that space the data of any image type can be converted into any other image type. The intermediate data space is described by a `COMMON_LINE` structure and can hold one line of image data (multi-channel is possible). This data is stored in either long integer format or in double precision floating point. The functions that puts the data in the `COMMON_LINE` (`conv_to_common()`) determines which data format is chosen, allocates the necessary data space and fills the `COMMON_LINE` structure to describe the data. The function of the receiving image type (`conv_from_common()`) must be prepared to handle both the integer and the floating point data. These functions are repeatedly called by the `convert()` function until all lines in the image are converted one by one. See "Data conversion (convert)" in chapter "Programming with Image" for more information.

The function `conv_to_common` should return `OK` (1) as long as not all lines have been written to the `COMMON_LINE` and `NOT_OK` (0) when it has written the last line. Its header:

```
void type_conv_to_common(IMAGE *im, int n,
                        COMMON_LINE *com_line )
```

Our byte 2d version :

```
int byte_2d_conv_to_common( IMAGE *im, int n, COMMON_LINE *com_line )
{
    BYTE_2D_IMAGE    *g_im;
    BYTE              *pix;
    long              *data;
    register long     npix;

    im_begin_func("byte_2d_conv_to_common");

    g_im = (BYTE_2D_IMAGE *) ImageIn(im);
    npix = Byte2dImageWidth(g_im);

    if( n == 0 ){          /* First time called */
        data = (long *) malloc( (size_t) npix * sizeof(long) );
        if(!data)
            return(im_report_error("byte_2d_conv_to_common", IE_NOMEM,
                                   "no memory allocated for data"));
        set_common_line(com_line, COM_LONG, data, 0, 0, 0, 0, 1, 0.0, 255.0);
    }

    data = com_line->data;
    com_line->x = npix;
    com_line->y = n;
    com_line->z = 0;

    pix = Byte2dImageData(g_im);
    pix += com_line->y * npix;

    while(--npix >= 0 )
        *data++ = *pix++;

    if( n == Byte2dImageHeight(g_im) - 1 ) {
        im_end_func("byte_2d_conv_to_common");
        return( 0 ); /* 0 = stop, this is the last line */
    }

    im_end_func("byte_2d_conv_to_common");
    return( 1 ); /* 1 = more lines are available */
}
```

The function `conv_from_common` should return OK (1) as long as it can store the data it is offered. If it can not store the data, it should return NOT_OK (0) which will stop the conversion. Its header :

```
int type_conv_from_common(IMAGE *im, int n,  
COMMON_LINE *com_line)
```

For the byte 2d image type this function does the trick :

```
int byte_2d_conv_from_common( IMAGE *im, int n, COMMON_LINE *com_line)
{
    BYTE_2D_IMAGE    *g_im;
    double            *d_ptr;
    long              *l_ptr;
    BYTE              *pix;
    int               type;
    register long     npix;
    register int      chan_offs;

    im_begin_func("byte_2d_conv_from_common");

    g_im = (BYTE_2D_IMAGE *) ImageOut(im);
```

```
type = com_line->type;
d_ptr = com_line->data;
l_ptr = com_line->data;

chan_offs = com_line->nr_channel;

npix = Byte2dImageWidth(g_im);
pix = Byte2dImageData( g_im );
pix += com_line->y * npix;

if( com_line->z == 0 ){
    if( type == COM_LONG ){
        while( --npix >= 0 ){
            *pix++ = *l_ptr;
            l_ptr += chan_offs;
        }
    }
    else if( type == COM_DOUBLE ){
        while( --npix >= 0 ){
            *pix++ = *d_ptr;
            d_ptr += chan_offs;
        }
    }
}
else if( type == COM_LONG ){
    while( --npix >= 0 ){
        if( *pix < *l_ptr )
            *pix = *l_ptr;
        pix++; l_ptr += chan_offs;
    }
}
else if( type == COM_DOUBLE ){
    while( --npix >= 0 ){
        if( *pix < *d_ptr )
            *pix = *d_ptr;
        pix++; d_ptr += chan_offs;
    }
}

if( com_line->t > 0 ) {
    im_report_error("byte_2d_conv_from_common", IE_NOT_OK, "");
    return( 0 ); /* 0 = stop, can't handle this */
}

im_end_func("byte_2d_conv_from_common");
return( 1 ); /* 1 = can handle more data */
}
```

The overload table

When all the basic functions have been created, an overload file must be created to tell the infrastructure which function to call for which basic service. "Overload tables" in the section "Programming with Image" describes the overloading mechanism and the format of the overload table. The source file 'overload.c' is generated from the overload tables as described in that section.

When the file 'overload.c' is compiled, the `type_ident` and `type_spec` of all valid image types must be known. The header file 'image.h' is included in 'overload.c' to provide

these definitions for the standard image types. The preprocessor directives '#include' and '#define' present in an overload file will be copied to the file 'overload.c'. By including the header file with these definitions in the overload file, the new image types are made known to the overload mechanism.

For the byte 2d image type construct a file 'byte_2d.ovl' with this contents (note the include directive '#include "byte_2d.h"):

```
#
#
# BYTE 2D OVERLOAD TABLE
#
#include "byte_2d.h"
#
TABLE byte2d      BYTE_2D          BYTE_2D_SPEC      2
#
# Basic functions for the byte_2d image type
#
    create_image          byte_2d_create_image
    destroy_image         byte_2d_destroy_image
    copy_part_image       byte_2d_copy_part_image
    copy_masked_part      byte_2d_copy_masked_part
    conv_to_common        byte_2d_conv_to_common
    conv_from_common      byte_2d_conv_from_common
    pix_value_str         byte_2d_pix_value_str
    get_image_window_info byte_2d_get_image_window_info
    part_image_display    byte_2d_part_image_display
#
```

Next, the basic functions as described here together with the low-level functions listed at the end of this chapter must be compiled and added to Image.

Low Levels

The prototypes of all the functions are located in a separate include file "byte_2dp.h"

```
#ifndef BYTE_2DP_H      /* necessary to avoid multiple definitions */
#define BYTE_2DP_H      /* file inclusion */

#include "image.h"
#include "byte_2d.h"
#include "roi.h"

/* prototypes of the basic functions */
#if defined(__STDC__) || !defined(INTERPRETED)
BYTE_2D_IMAGE *byte_2d_create_image(char *, int, int, int, int);
void byte_2d_destroy_image(BYTE_2D_IMAGE *);
int byte_2d_copy_part_image(IMAGE *, IMAGE *, int, int, int, int, int, int,
int, int, int);
int byte_2d_copy_masked_part(IMAGE *, BOOL_MASK *, IMAGE *, int, int, int,
int, int, int, int, int, int, int);
char *byte_2d_pix_value_str(IMAGE *, int, int, int);
void byte_2d_get_image_window_info(IMAGE *, char []);
int byte_2d_conv_to_common( IMAGE *, int, COMMON_LINE *);
int byte_2d_conv_from_common( IMAGE *, int, COMMON_LINE *);

BYTE *l_byte_read_part_image(BYTE *, int, int, int, int, int, int, int,
int, int);
void l_byte_write_part_image(BYTE *, BYTE *, int, int, int, int, int, int,
int, int, int);
```

```
BYTE *l_byte_read_masked_part(BYTE *, BOOL_MASK *, int, int, int, int, int,
int, int, int, int);
void l_byte_write_masked_part(BYTE *, BYTE *, BOOL_MASK *, int, int, int,
int, int, int, int, int, int, int);
#endif
#endif /* BYTE_2DP_H */
```

The following is a source code listing of the low-level functions used in the sample code in this section .

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "image.h"
#include "im_error.h"
#include "im_infra.h"
#include "roi.h"

#include "byte_2d.h"
#include "byte_2dp.h"

BYTE *l_byte_read_part_image(BYTE *ip, int sx, int sy, int sz,
                             int width, int height, int depth,
                             int sw, int sh, int sd)
{
    BYTE *sp, *dp, *ptr;
    int y, z;

    im_begin_func("l_byte_read_part_image");

    if (!(ptr = (BYTE *) malloc ((size_t)( (long) width * height *
                                         depth * sizeof(BYTE)))) {
        im_report_error("l_byte_read_part_image", IE_NOMEM,
                       "couldn't allocate memory for buffer");
        return( NULL );
    }

    for (z=0 ; z<depth; z++) {
        for (y=0 ; y<height ; y++) {
            sp = ip + ((z+sz) * sw * sh) + ((sy+y) * sw) + sx;
            dp = ptr + (z*width*height) + (y*width);
            memcpy(dp, sp, width * sizeof(BYTE));
        }
    }
    im_end_func("l_byte_read_part_image");
    return( ptr );
}

void l_byte_write_part_image(BYTE *ip, BYTE *data,
                             int dx, int dy, int dz,
                             int width, int height, int depth,
                             int dw, int dh, int dd)
{
    BYTE *dp, *sp;
    int y, z;

    im_begin_func("l_byte_write_part_image");

    for (z=0 ; z<depth; z++) {
        for (y=0 ; y<height ; y++) {
            dp = ip + ((z+dz) * dw * dh) + ((dy+y) * dw) + dx;
            sp = data + (z*width*height) + (y*width);
            memcpy(dp, sp, width * sizeof(BYTE));
        }
    }
}
```

```

    }
    im_end_func("l_byte_write_part_image");
    return;
}

BYTE *l_byte_read_masked_part(BYTE *ip, BOOL_MASK *mask,
                             int sx, int sy, int sz,
                             int width, int height, int depth,
                             int sw, int sh, int sd)
{
    BYTE          *sp, *dp, *ptr;
    register int  x, y, z;
    BYTE          *mp, byte_mask;
    int           line_offset;

    im_begin_func("l_byte_read_masked_part");

    line_offset = ((width - 1) / 8) + 1;
    if (!(ptr = (BYTE *) calloc ( (size_t) sizeof(BYTE),
                                (size_t) width * height * depth ))) {
        im_report_error("l_byte_read_masked_part", IE_NOMEM,
                       "couldn't allocate memory for buffer");
        return( NULL );
    }

    dp = ptr;
    for (z=0 ; z<depth; z++) {
        for (y=0 ; y<height ; y++) {
            sp = ip + ((z+sz) * sw * sh) + ((sy+y) * sw) + sx;
            mp = BoolMaskData(mask) + z*height*line_offset +
                y * line_offset;
            byte_mask = 0x80;
            for (x=0 ; x<width ; x++) {
                if ( *mp & byte_mask )
                    *dp++ = *sp++;
                else {
                    dp++;
                    sp++;
                }
                if (!(byte_mask >>= 1)) {
                    mp++;
                    byte_mask = 0x80;
                }
            }
        }
    }
    im_end_func("l_byte_read_masked_part");
    return( ptr );
}

void l_byte_write_masked_part(BYTE *ip, BYTE *data, BOOL_MASK *mask,
                              int dx, int dy, int dz,
                              int width, int height, int depth,
                              int dw, int dh, int dd, int clear)
{
    BYTE          *dp, *sp;
    BYTE          *mp, byte_mask;
    int           x, y, z, line_offset;

    im_begin_func("l_byte_write_masked_part");

    line_offset = ((width - 1) / 8) + 1;

    sp = data;
    for (z=0 ; z<depth; z++) {
        for (y=0 ; y<height ; y++) {

```

```
    dp = ip + ((z+dz) * dw*dh) + ((dy+y) * dw) + dx;
    mp = BoolMaskData(mask) + z*height*line_offset +
        y * line_offset;
    byte_mask = 0x80;
    for (x=0 ; x<width ; x++) {
        if ( *mp & byte_mask )
            *dp++ = *sp++;
        else {
            if (clear)
                *dp++ = 0;
            else    dp++;
            sp++;
        }
        if (!(byte_mask >>= 1)) {
            mp++;
            byte_mask = 0x80;
        }
    }
}
im_end_func("l_byte_write_masked_part");
return;
}
```

Testing the image type

If the above steps have been followed correctly, a new version of Image has been compiled and linked which includes the byte_2d image type. The following sample code can be used to test the new image type. Please note that this is only a code-fragment, not a complete program, it expects that the Image library is has been initialized.

```
#include "image.h"
#include "im_error.h"
#include "im_infra.h"
#include "roi.h"
#include "byte_2d.h"

IMAGE *im_a, *im_b, *im_c, *im_d;
IMAGE *roil;
BOOL_MASK *bm;

/*
 * create two grey images, read a file from disk, convert it to and
 * then from the new type and save it to a file again to be able to
 * compare it with the original image data
 *
 * FUNCTIONS USED: create_image, conv_to_common, conv_from_common
 */
im_a = create_image("A", GREY_2D, 256, 256,1);
readfile("orka256",im_a, 0,0);
im_b = create_image("B", GREY_2D, 256, 256,1);
convert(im_a, im_b, BYTE_2D);
convert(im_b, im_a, GREY_2D);
writefile(im_a,"result1", ICS_F);

/*
 * create image of new type, copy a part of the previously used
 * images to it. To check the result, convert it back to grey, and
 * save it to disk.
 *
 * FUNCTIONS USED: create_image, copy_part_image, conv_to_common
 */
```

```
im_c = create_image("C", BYTE_2D, 256,256, 1);
copy_part_image(im_b, im_c, 64, 64, 0, 128, 128, 1, 0,0,0);
convert(im_c, im_a, GREY_2D);
writefile(im_a, "result2", ICS_F);

/*
 * create a Boolean mask from a grey_2d image, convert the grey
 * image to the new type. Define a roi in the new type with a
 * Boolean mask, and convert that the data from that ROI to a
 * different image type.
 *
 * FUNCTIONS USED: create_image, conv_from_common, destroy_image,
 *                 copy_masked_part
 */
readfile("trui", im_a, 0,0);
threshold(im_a, im_b, 128);
bm = get_bool_mask(im_b);
convert(im_a, im_c, BYTE_2D);
im_d = create_image("D", BYTE_2D, 256, 256, 1);
roil = roi_define("roil", im_c, 0,0,0, 256, 256, 1, bm);
convert(roil, im_d, FLOAT_2D);
writefile(im_d, "result3", ICS_F);
```

We have described a very limited implementation of the byte image type. To do something useful with the new image type, additional image processing operations should be added in a similar way.

Defining an image subtype

Image subtypes can be implemented for a number of reasons. The difference in the interpretation of the image data is one reason, but sub-typing can also be used as an organizational tool. It is possible to design a set of operations that is targeted to a specific processing domain such as, microscopical images. By limiting the operations that can be performed on specific images, their specialized nature can be emphasized and the user can be aided in processing them.

The implementation of an image subtype is similar to that of a new image main type described in the previous section. Some definitions need to be made and the service functions must be made available. The difference lies in the fact that an image subtype can make use of functions already implemented for another image type. In this section, we describe how to implement an image subtype. The standard Image Library image-type LABEL_2D is a subtype of the GREY_2D image type and will therefore be used as an example.

First some rules and guide-lines:

- 1) The sub-type **must** have an unique `type_ident`. The value of the `type_ids` of the standard image types are listed in the header file 'image.h'. The values 1 to 32 are reserved for main types and should not be used for a subtypes. See the difference between LABEL_2D and GREY_2D, GREY_2D being a main type is in the range 1 to 32 and the subtype LABEL_2D is 33.
- 2) The `type_spec` of the subtype **must** have the same value as the `type_spec` of the main type. A new define is recommended for readability of the source code. In 'image.h' the `type_spec` L_2D_SPEC (of the label_2d image type) is defined to be equal to G_2D_SPEC (the `type_spec` of grey_2d).
- 3) In order to make use of the functions of the main type, the type descriptor should be the same as that of the main type. The subtype can then use the functions from the existing image type. In the header file 'label_2d.h' the `type_descriptor` LABEL_2D_IMAGE is defined as the type descriptor GREY_2D_IMAGE. A new name and new macros are defined to be used in the source code for this subtype.
- 4) If additional fields in the type descriptor are necessary, copy the structure of the main type and add the extra fields at the end. It should be realized that the functions that are inherited from the main type do not know about these fields and therefore do not initialize them or operate on them in any way. In most cases, this requires that the service functions will have to be re-implemented for the subtype. The service functions are described in "Implementing a new image type".

The defines of the `type_ident`, the `type_spec`, the type descriptor and the macros to access the fields of the descriptor should be put in a separate header file named after the image type. In the case of the label_2d subtype the definition of the `type_ident` and the `type_spec` is located in 'image.h' and not in 'label_2d.h'.

As with a new image type, an overload file must be created to tell the infrastructure which functions to call for this image type. Looking at the overload file of the label_2d image type, we can see that most of the service functions are inherited from the grey_2d image main type:

```
# LABEL 2D OVERLOAD TABLE
#
# NOTE LABEL 2D INHERITS MOSTLY FROM GREY_2D
#

TABLE      12d      LABEL_2D      L_2D_SPEC      2

#
# Basic functions for each image type
#
    create_image          g_2d_create_image
    destroy_image         g_2d_destroy_image
    copy_part_image       g_copy_part_image
    copy_masked_part      g_copy_masked_part
    conv_to_common        g_2d_conv_to_common
    conv_from_common      g_2d_conv_from_common
```

```
part_image_display      l_2d_part_image_display
pix_value_str           g_2d_pix_value_str
get_image_window_info   l_2d_get_image_window_info
#
```

As with implementing a new image type, the last three functions from the list above are only required when defining an image subtype in the complete `SCIL_Image` environment.

The dedicated service functions and the overload file must be added to `Image` as described in previous sections. The image subtype is then implemented and can be used. Further image processing operations can be added either by inheritance from the main type or by adding a new operation. Inheriting an operation from the main type is done by copying the entry in the overload table from the main type to the overload table of the subtype.

Chapter 13 Creating an Image Application

This chapter discusses the topic of creating a stand-alone application using the Image library.

Read this chapter if :

- you want to build an application with the Image Library

Introduction

The Image Library can be used to build your own image-processing application. The library can best be regarded as an application without a front-end. Whenever a function is called in the library, the behavior and state of entire library may be affected by it. To use the Image library, only one (obligatory) step has to be taken. That is:

- Initialization of the library before any image processing function is called. See "Initialization" on page 13-2.

Once this requirement is satisfied, the conditions are set for proper execution of the image processing functions of the Image library. When coding an application, several topics may need additional attention:

- What to do when an error occurs in the Image library and how to retrieve information about the error situation. See "Error handling" on page 13-2.
- How to redirect text output from the library to the desired location. See "Text output handling" on page 13-3.

Initialization

Before any image processing function can be called, the library must be initialized. The function that performs all the necessary actions is `initimage()`. Its prototype:

```
void initimage( void );
```

Although for a complete application additional steps are necessary to get automatic notification of all kinds of "events" such as errors, creation of images, changes to images etc. the `initimage()` function is sufficient to get the Image library running correctly.

Error handling

In the event of an error occurring in the application (due to whatever reason), the program must detect the situation and handle it appropriately. Either by trying to correct to problem or quitting the application. For correct assessment of the situation, sufficient information is required. As discussed in "**Error handling and reporting**" on page 9-21, the Image library logs the location and nature of an error occurring in the library in detail. After a processing-

function has handled an error by cleaning up and returning with a fault status, the application can then review the information.

Text output handling

The default behavior of the Image library is to show all text output through the `stdout` and `stderr` streams. An application can overrule this by supplying a function pointer to which the text must be redirected, see "**Textual output**" on page 9-26.

Color Lookup tables.

A principal feature of the user interface of an image processing application is the display of the images. Often color lookup tables are used to influence the representation of the images on the screen, either to enhance the contrast or highlight specific features in the image. Because the Image Library already supplies color lookup tables, it is a good idea for the display interface to also make use of these tables instead of defining a new set.

The Image Library defines that when the contents of a clut is changed by an operation, this must be published through the `super_clut` object. The display interface is then able to react to the changed CLUT providing it subscribed to the `super_clut` object. For additional information on cluts, see "Image Color Lookup Tables" on page 9-11.

Sample code

The Image library is accompanied by a sample program to show some of the important issues involved in using the library. In the listing below we have numbered these issues and they are discussed in more detail.

The actual image-processing done is limited, an image is read, thresholded, labeled and some features of all the objects in the image are measured and printed.

1. The pointer to the top-level image is retrieved, this object publishes the creation and destruction of images as discussed in "Publish and Subscribe in the Image library" on page 8-5.
2. The pointer to the error-stack is retrieved. This stack can be used to get detailed information on an error occurring in the library. More information on the error-stack can be found in "Error handling and reporting" on page 9-21.

3. To redirect text-output, a function can be registered. Details of which can be found in "Textual output" on page 9-26. The function itself analyses the `stream` parameter and if it is a special stream, it prints a header above the text indicating which stream.
4. The `im_handle_error()` function is subscribed to the error-stack. When an error occurs and the library returns, this function is automatically called and prints a stack-trace of the location of the error.
5. The `handle_super_im()` function is subscribed to the top-level image. When an image is created, this function prints a message that a new image is created and then subscribes the `handle_images()` to the new image. `handle_images()` in its turn will print out information about the publishes that come from these images.
6. `initimage()` initializes all the necessary internal tables of the library. It is very important that the initialization is done otherwise most image-processing functions will not operate at all.
7. Finally the image processing itself, in this sample we choose to leave out checking the return-values of each function except for the initial reading of an image. In a real application, additional statements are needed to check that each operation performed its task correctly and if not, take appropriate action.

```
#include <stdio.h>
#include <string.h>
#include "support.h"
#include "spublish.h"
#include "image.h"
#include "im_error.h"
#include "generic.h"
#include "im_infra.h"
#include "imonly.h"
#include "imtxtout.h"
#include "im_aio.h"

static char errbuf[2048]; /* local buffer for printing etc. */
IM_ERROR_STATUS *err_stack;
void *super_image = NULL;

int main(int argc, char *argv[])
{
    IMAGE *im, *im1, *im2;
    LIST *l, *o;
    OBJECT *obj;

    /* get the global objects to subscribe to */
    super_image = get_super_im(); /* #1 */
    err_stack = get_im_error_stack(); /* #2 */

    /* set our own text-output handler */
    im_set_output_handler( my_output_func ); /* #3 */

    /* subscribe to the error-stack */
    spb_subscribe( err_stack, NULL, im_handle_error, 0L ); /* #4 */

    /* subscribe to image-class (super_im) */
```

```
    spb_subscribe( super_image, NULL, handle_super_im, 0L );    /* #5 */

/* initialize the image library */
initimage();    /* #6 */

im = readfile("cermet",NULL,0,0);
if ( !im ) {    /* #7 */
    image_output(IMO_OUTPUT, "Can not find image, exiting!!\n");
    return(0);
}

im1 = create_image("im1", BINARY_2D, 256, 256, 1);
threshold(im,im1,128);
invert_im(im1,im1);
eval("cermet=255-cermet",0);

im2 = create_image("im2", GREY_2D, 256, 256, 1);
l = list_label(im1,im2, 8, 20);

ForAllElements(o,l) {
    obj = (OBJECT *)Info(o);
    image_output(IMO_OUTPUT, "Label: %4ld,",obj->label);
    object_shape_meas(im2, o, AREA|PERI|CR );
    image_output(IMO_OUTPUT," Area: %6ld, Peri: %8g, Cr: %8g\n",
        area(o), peri(o), cr(o));
}

return (IE_OK );
}

/*
 * This function retrieves the information from the error-stack
 * and puts it in a textbuffer.
 */
void my_dump_err_stack(IM_ERROR_STATUS *imerr, char *errbuf)
{
    int i;
    char *ptr, *fptr;
    char *ep;

    sprintf(errbuf,
        "Image error stack-trace\n===== \n\n");
    ep = errbuf + strlen( errbuf );
    for(i=1; i<=imerr->edepth; i++ ) {    /* for each level */
        /* pointer to message */
        ptr = imerr->mstack + imerr->moffs[i-1];
        /* pointer to funcname */
        fptr = imerr->fstack.names + imerr->fstack.noffs[i-1];
        /* print funcname and error-status */
        sprintf(ep, "%s : [%d];", fptr, imerr->sstack[i-1]);
        ep = errbuf + strlen( errbuf );

        /* if error message, print it */
        if ( ptr && (*ptr) ) {
            sprintf(ep, "\"%s\"",ptr);
            ep = errbuf + strlen( errbuf );
        }
        /* if recursive, print recursion depth */
        if ( imerr->fstack.rcount[i-1] > 1 )
            sprintf(ep, " ; rec_depth = %d\n",
                imerr->fstack.rcount[i-1]);
        else
            sprintf(ep, "\n");
        ep = errbuf + strlen( errbuf );
    }
}
```

```
    }
    return;
}

/*
 * Function: im_handle_error
 *
 * This function is called when an error has occurred and has been handled
 * by the image library. In the main loop it is subscribed to the error
 * stack.
 */
void im_handle_error(IM_ERROR_STATUS *im_error, void *dummy, int mess, int
*data, int *cl)
{
    IM_ERROR_STATUS imerr;

    imerr = *im_error;
    my_dump_err_stack(&imerr, errbuf); /* print the info to a buffer */
    image_output(IMO_ERROR, errbuf); /* print the buffer */
    return;
}

/*
 * This function handles the text-output from the image_output function.
 */
void WINAPI my_output_func( int stream, char *text )
{
    switch ( stream ) {
        case IMO_INSTRUCT:
            printf("My Image Instruction:\n-----\n");
            printf( text );
            break;
        case IMO_WARNING:
            printf("My Image Warning:\n=====\n");
            printf( text );
            break;
        case IMO_ERROR:
            printf("My Image Error:\n=====\n");
            fprintf(stderr, text);
            break;
        case IMO_OUTPUT:
            default:
                printf( text );
                break;
    }
    return;
}

/*
 * This function is subscribed to each new image by handle_super_im()
 * and prints information about publishes from this image.
 */
void handle_images(void *image, void *id, int mess, void *data, void *cl)
{
    spbAREA3D *im_area;

    image_output(IMO_OUTPUT, "Image Handler : message = %d, data = %d\n",
mess, data);
    if ( is_image(image) )
        image_output(IMO_OUTPUT, "Imagepointer to %s\n",
ImageName((IMAGE*)image));

    switch ( mess ) {
        case SPB_CHANGED :
            if ( im_area = data ) {
                image_output(IMO_OUTPUT,
```

```
        "Rectangle changed: %d %d %d %d %d %d\n",
        im_area ->x, im_area ->y, im_area ->z,
        im_area ->width, im_area ->height,
        im_area ->depth);
    }
    break;
default :
    break;
}
}

/*
 * This function detects if new images have been made and
 * subscribes the handle_images function to each new image
 */
void handle_super_im(void *sup_im, void *id, int mess,
void *data, void *cl)
{
    image_output(IMO_OUTPUT, "Super Image : message = %d, data = %d\n",
        mess, data);
    if ( is_image(data) ) {
        image_output(IMO_OUTPUT, "Data is Imagepointer to %s\n",
            ImageName((IMAGE*)data));
    }

    switch ( mess ) {
        case SPB_ITEM_ADD :
            /* subscribe to each new image */
            spb_subscribe( data, NULL, handle_images, 0L);
            break;
        case SPB_ITEM_DELETE :
            /* unsubscribe from images that are destroyed */
            spb_unsubscribe( data, NULL, handle_images);
            break;
        default :
            break;
    }
}
}
```

