Programming SURVO 84 in C

Seppo Mustonen

University of Helsinki
Department of Statistics
1989
SURVO 84C is an integrated system for statistical analysis, computing, data base management, graphics, desktop publishing, etc. Through its unique editorial interface, SURVO 84C forms a general environment for many kinds of applications.

SURVO 84C Contributions is a series of papers devoted to various new features of the SURVO 84C system.

Editor
Seppo Mustonen
University of Helsinki, Department of Statistics
Aleksanterininkatu 7, 00100 Helsinki, Finland

Copyright © 1989 by the author
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author.

The contents of this paper are furnished for informational use only, are subject to change without notice, and should not be construed as a commitment by the author. The author assumes no responsibility or liability for any errors or inaccuracies that may appear in this paper. The software described in this paper is furnished under license and may be used or copied only in accordance with the terms of this license.

This paper was composed and written using SURVO 84C. The original of the paper was created as a PostScript file by the PRINT operation of SURVO 84C. Proofs were printed on the QMS-PS 810 PostScript printer and the final camera-ready copy was set on a Linotype 300 typesetter at PrePress Studio, Helsinki. The final copies were printed from the camera-ready copy by University Press, Helsinki, Finland.

ISSN 0786-2792
SURVO 84C Contributions

1. S.Mustonen: PostScript printing in SURVO 84C, 1988
2. S.Mustonen: Sucros in SURVO 84C, 1988
Seppo Mustonen
Programming SURVO 84 in C

1. Introduction 1
2. SURVO 84C processes 3
3. Example of a SURVO 84C module 5
4. Edit field 15
5. Shadow lines 17
6. Space allocation 19
7. Include files 20
8. Libraries 21
  8.1 SURVO.LIB 22
  8.2 SURVOMAT.LIB 67
  8.3 DISTRIB.LIB 79
Index to SURVO 84C library functions 85

Printed in Finland 1989
Programming SURVO 84 in C

Seppo Mustonen
Department of Statistics, University of Helsinki, Finland

Abstract: The SURVO 84C system can be extended without any limits by new program modules written in C. In this paper, the program structure of SURVO 84C is described. Instructions for making program modules are given. The tools developed for SURVO 84C programming are presented as C library functions.

Keywords: SURVO 84C, C language, Programming tools

1. Introduction

SURVO 84C is an open system. It provides tools for making extensions in different ways. Many of the enhancements can be accomplished by means of sucros described in "Sucros in SURVO 84C" by Mustonen (1988). Also the matrix interpreter (MATRUN operations), the touch mode and the editorial computing mode, for example, are useful when making such extensions. Furthermore, some of the general SURVO 84C operations (like PRINT and PLOT) provide their own special programming tools.

In the most demanding tasks, the only general way for making extensions is to use the programming language C since the entire SURVO 84C system has been programmed in C.

SURVO 84C is a modular system consisting of one main program (the editor) and of numerous modules which are called by the main program when needed. Each module itself is written like any independent program, but it can be executed only as a child process when its parent process (main program) is present and delivers all input information to the module. When a SURVO 84C module terminates, it disappears from the memory and the control returns to the main program. It is up to the module how it renders its results. Usually results are printed on the screen and saved in an
output file or they are displayed in the edit field. Each module in turn may have children which are written independently but receive sustenance from their mother.

An important link between the main program and the modules (and thus also between different modules) is the edit field, which among other things carries vital input information to the modules. The main program selects the module needed for each particular task according to the activation procedure initiated by the user. In most cases the activated line in the edit field holds that information. For example, when the user activates a line starting with the word PRINT, the main program ‘knows’ that a program !PRINT.EXE should do the job and calls it immediately. Thereafter it is the task of !PRINT.EXE to read parameters and other input information (like specifications written around the activated line).

This convention guarantees that there can be as many modules in the system as there are permitted program file names (starting with ‘!’). Since each module visits the central memory in turn, it is only the disk capacity that may limit the size of the system.

Instead of one large program, we have a family of smaller programs which can cooperate. Such a system is easier to maintain. For example, each module usually consists of several compilands which are compiled separately and then linked together. There is no need to link the modules to each other or to the main program. However, the environment created for the programmer guarantees that making a SURVO 84C module is like extending one large uniform system which could be one huge program.

When programming a new module, it is not necessary to know about the requirements of other modules (assuming that we are not using conflicting names for modules). In many cases, however, it is good to be familiar with other solutions and use ready-made tools generated earlier for similar purposes.

In fact, programming is highly simplified when various tools which have been developed earlier are employed whenever possible. The standard tools of SURVO 84C are available as libraries. The main program and all existing modules have been written in the C language. In principle, any other language producing executable program files (like Pascal, Fortran, Assembler) could be used as well, but for the time being they are lacking the SURVO 84C library support.

The main purpose of this paper is to provide information for those people who would like to make more SURVO 84C modules. We give some rules which should observed and a great deal of recommendations. Finally, we describe the tools available and give examples of their use.
Although anyone who writing a SURVO 84C compatible program can select the tools as he/she wishes, there are clear advantages to following the recommendations. Generally adopted tools create a common style in the system structure that in many ways helps the user. For example, when the user tries to test a new operation of SURVO 84C, he has the right to expect it to work according to patterns encountered earlier in similar operations. The parts of the SURVO 84C ‘world’ should resemble each other as much as possible, at least formally. This increases the confidence of the user to the system. On the other hand, we don’t wish to spoil the joy of inventing new approaches. As an open system SURVO 84C will permit and tolerate several alternative solutions in any application area.

The prerequisites for a SURVOC 84C programmer are that he/she is able to use the C language and knows the idea and basic solutions of SURVO 84C from the user’s point of view.

The current technical requirements are the *Microsoft C Compiler* (Ver. 5.10 or newer), the *SURVO 84C libraries* and the *SURVO 84C system itself*. The present implementation of SURVO 84C is for the MS-DOS operating system. Due to the origin and portability of the C language, it is obvious that versions for Unix-like operating systems at least are not difficult to develop.

2. SURVO 84C processes

The term process is described in the Microsoft C run-time library reference as follows (p.73):

*The term “process” refers to a program being executed by the operating system. A process consists of the program’s code and data, plus information pertaining to the status of the process, such as the number of open files. Whenever you execute a program at the MS-DOS level, you start a process. In addition you can start, stop, and manage processes from within a program by using the process control routines.*

The possibility to start up another process during the program as a ‘child’ process is crucial in the construction of SURVO 84C. There are a few alternatives for calling child processes. The new process may overlay the parent process or the parent process may stay resident during the child’s execution. Both alternatives are used in SURVO 84C, and in most cases the latter one, since the main program remains always resident until the end of the session.

As a consequence of this construction principle, we can always call other programs easily while staying in SURVO 84C in the same way as SURVO 84C calls its children. The only provision is that there is
enough memory left for the new process and it can be accessed from the current directory (of SURVO 84C). Thus all MS-DOS commands may be given like any SURVO 84C command directly from the edit field by putting the ‘prompt’ symbol ‘>’ before the command and by activating it like a SURVO 84C operation. For example, \texttt{>DIR A:*.EDT} lists all edit files on disk \texttt{A:}.

Similarly we can start any executable program (.EXE or .COM) or batch file (.BAT) during the SURVO 84C session. For example, \texttt{>S} always starts a new SURVO 84C copy as a child of the current one (since S.EXE is the main program of SURVO 84C). Upon returning from child S we are back in the original SURVO 84C session.

Hence most programs without modifications may serve the SURVO 84C system as its child processes. This is very helpful for experienced users, since they can employ SURVO 84C as a natural extension of the operating system and do everything while staying in SURVO 84C.

However, to make a program a true SURVO 84C module some considerations related to input and output should be taken into account. Also the general requirements and the style of SURVO 84C programming may imply modifications in existing programs.

A formal distinction between a SURVO 84C module and a general program is that the file names of directly callable SURVO 84C modules start with ‘!!’. Furthermore, the SURVO 84C modules receive all the input information directly from the main program (editor) so that they cannot be executed alone.

The link between the main program and a module is one address given by the main program as a parameter and pointing to an array of pointers. This array tells the addresses of the SURVO 84C system parameters and variables so that the module may use the same information as the main program does. Then, from the programmer’s point of view, the module is an integrated part of the main program. For real access to those parameters and variables, the module has to call first an initiation function (\texttt{s_init}).

During its work, the module may update various system variables (for example, write results in the edit field) so that the effects of the module can be seen immediately after returning to the main program.

The cooperation between the main program and the modules strengthens the system. The system is more than a collection of different programs. Therefore it is important to take full advantage of these possibilities for interaction when creating new modules.
3. Example of a SURVO 84C module

The idea and practice of making SURVO 84C modules is first illustrated by an example. To save space and to highlight the main principles, we shall describe coding of a simple module for calculating weighted means from statistical data.

Usually it is good to start by making a synopsis from the user’s point of view and imagine how the things should look if we already had the new operation. In this case we could type following text in the edit field:

Here we have a small application where the data set is on edit lines 4-12, the MEAN operation on line 17 and results (which we hope to receive after activation of the MEAN line) on lines 19-22.

We assume that the MEAN operation has the following syntax:

**MEAN <SURVO_84C_data>,<first_line_for_the_results>**

To select variables and observations, we have used two extra specifications (on lines 14-15). There **MASK=--AAW** selects only columns #3 and #4 (**Test1,** **Test2**) for the analysis and column #5 (**Test3**) is used as a weight variable. **CASES=Sex:M** indicates that only observations with **Sex=M** are selected.

We shall see that there will be still more options available if the MEAN module is written according to the standards of SURVO 84C, and all this is achieved with a minimal effort by using ready-made tools of the SURVO 84C libraries.
It should also be noted that the structure of more complicated modules does not differ from that of this example.

The !MEAN module has only one compiland and its main function is listed below in several parts. The line numbers have been added for easier reference.

```c
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <malloc.h>
#include "survo.h"
#include "survoext.h"
#include "survodat.h"

SURVO_DATA d;
double *sum;       /* sums of active variables */
long   *f;         /* frequencies */
double *w;         /* sums of weights */
long n;
int weight_variable;
int results_line;

main(argc,argv)
int argc; char *argv[];
{
    int i;

    if (argc==1)
    {
        printf("This program can be used as a SURVO 84C module only.");
        return;
    }
    results_line=0;
    if (g>2)
    {
        results_line=edline2(word[2],1,1);
        if (results_line==0) return;
    }
    i=data_open(word[1],&d); if (i<0) return;
    i=sp_init(r1+r-1); if (i<0) return;
    i=mask(&d); if (i<0) return;
    weight_variable=activated(&d,'W');
    i=test_scaletypes(); if (i<0) return;  /* permitted only once */
    i=space_allocation(); if (i<0) return;
    compute_sums();
    printout();
    free(sum); free(f); free(w);
    data_close(&d);
}
```

Among the include lines, 8-10 refer to special SURVO 84C include
files. Lines 8-9 should always be present in modules. Line 10 (survodat.h) is needed especially in those modules where SURVO 84C data sets and data files are employed.

Line 12 declares the SURVO_DATA structure d which may represent either a data set in the edit field (as DATA TEST in our example) or a SURVO 84C data file or part of it or even a matrix file. The writer of the module has no need to know the actual form of the data set. By using the tools provided by the SURVO 84C library (like data_open on line 48), all these alternatives can be handled similarly. In rare cases where a distinction has to be made, the d.type member of the SURVO_DATA structure d gives the type of the data set at hand.

On lines 13-15, pointers to various arrays used in MEAN are declared. In order to make the modules general and flexible, we avoid fixed limits in arrays. Therefore all arrays whose sizes depend on application (like number of variables in the analysis) should be defined dynamically. This is done by using the standard space allocation function malloc. It has been employed here for all space reservations through the space_allocation call on line 54.

Finally, before the main function starts, certain global variables are declared on lines 17-19. To shorten the function calls, we usually prefer using static variables.

When calling the !MEAN module as a child process, the main program of SURVO 84C passes only one parameter (address of the pointer to the array of system pointers as a string). In the main function of !MEAN this parameter (argv[1]) is needed in the s_init call (line 31). It declares all important SURVO 84C system parameters and variables for !MEAN. Thereafter writing of code in !MEAN is like making more functions for the main program.

However, before the s_init call, lines 26-30 are given in order to prevent misuse of !MEAN (direct call of !MEAN from the MS-DOS level).

After the s_init call we have, for example, r=current line on the screen and r1=first visible edit line on the screen. Hence r1+r-1 is the current (activated) edit line. See the library reference of s_init for the complete list of system variables which are initialized by s_init.

The s_init function also analyzes the edit line (MEAN TEST,19) which was activated by the user and splits it into parts word[0]="MEAN", word[1]="TEST" and word[2]="19" giving the total number of ‘words’ found as g. (In this case g=3).

Lines 32-41 are for testing the completeness of the user’s call. Observe
that **MEAN** TEST without an edit line for the results is allowed and thus only the case \( g<2 \) (mere **MEAN** activated) leads to an error message.

In such a case, the standard modules typically give a short notice of their usage like "**Usage: MEAN <data>, L**" and the user can get more information by consulting the inquiry system of SURVO 84C.

On a new module written by the user, the inquiry system cannot provide any information. Therefore it is important to give longer explanations telling all essential features. This should be done with functions `initRemarks`, `rem_pr`, and `waitRemarks` as shown on lines 32-41. These functions emulate the behaviour of the inquiry system. For example, the user can load the explanations appearing on the screen to the edit field.

The next section in the main function (lines 42-47) deals with output in the edit field. As pointed out earlier, the line label (or number) for the results in the edit field may be omitted (case `resultsLine=0`). If the line for the results is given (i.e. \( g>2 \)), it is found by the SURVO 84C library function `edline2` (line 45). If no edit line corresponding to the user’s command is found, `edline2` gives an error message and returns 0 instead of the line number.

Line 48 `i=data_open(word[1],&d); if (i<0) return;` opens the data set and initializes several variables (members of structure `SURVO_DATA` `d`) describing the size and the structure of the data set. For example, we have the following information readily available for the subsequent processing:

- `d.m` # of variables in data (type int)
- `d.m_act` # of active variables (int)
- `d.n` # of observations in data (long)
- `d.l1` first active observation (long)
- `d.l2` last active observation (long)
- `d.varname[0],...,d.varname[d.m-1]` names of variables (char **)
- `d.vartype[0],...,d.vartype[d.m-1]` types of variables (char **)
- `d.v[0],...,d.v[d.m_act-1]` indices of the active variables (int *)

If the data is not available, `data_open` displays an error message and
returns -1. In that case there is an immediate return to the main program of SURVO 84C.

In SURVO 84C, the operations are not only controlled by parameters written on the activated line (like TEST and 19 in our example), but the modules can also be guided by using various specifications written around the activated line anywhere in the edit field. In our example, such specifications are \texttt{MASK=-AAW} and \texttt{CASES=Sex:M}.

To take their effects into consideration, we must first read all the specifications written in the current edit field. This happens by calling the \texttt{ssp_init} function once (line 49: \texttt{ssp_init(r1+r-1);p_init(r1+r-1);}) where the argument refers to the line currently activated. It implies \texttt{ssp_init} to look for specifications primarily around that line. Later the \texttt{spfind} function is called repeatedly to find specifications from a list generated by \texttt{ssp_init}.

The \texttt{mask} function (on line 50) has the task of analysing the VARS specification (or if it does not appear, the MASK specification) through the \texttt{spfind} function. If VARS or MASK exists, \texttt{mask} corrects the activation status of each variable accordingly. If VARS (MASK) is not given, the status of the data set itself determines which are active variables.

Line 51 checks whether any of the variables in the data set have been activated by ‘\texttt{W}’ (using the \texttt{activated} function). If such a variable is found (as \texttt{TTest3} in our example) the index of that variable is returned and it serves as a weight variable in the computations. Otherwise \texttt{activated} returns -1.

One of the unique features of SURVO 84C is the possibility to assess the validity of various statistical methods by checking the scale types of variables. Scale types can be declared for variables in data files only. The user has the freedom to use or not to use this facility. The \texttt{ttest_scaletypes} call on line 52 does the job in a positive case.

The observations may be restricted by the CASES and IND specifications. The \texttt{conditions} function (called on line 53) tests that those specifications, if used at all, are written correctly and initializes system variables which are used for scanning data during the computation (through a function called \texttt{unsuitable}).

After these preliminary checks, we are ready to allocate space for frequencies, sums of weights and weighted sums of observations. The dimension of these arrays must be \texttt{d.m_act}. This happens by calling \texttt{space_allocation} (line 54).

If the space is successfully allocated (there is no negative response), the actual computations can start (\texttt{compute_sums}) and the results are printed (\texttt{printout}).
Finally (on lines 57-58), the allocated space is freed and the data set closed before returning to the main program of SURVO 84C and to the normal editing mode.

Most of the functions called by the main function of !MEAN are either in the Microsoft C run-time library or in the SURVO 84C libraries. The descriptions of the SURVO 84C library functions will be given later in this paper.

There are only 4 functions called in the main function being specific for the !MEAN module, namely ttest_scaletypes, space_allocation, compute_sums, and printout. Since !MEAN is a very small module, all of them are in the same compiland together with the main function.

The ttest_scaletypes function has the following form:

```c
61 test_scaletypes()
62 { int i, scale_error;
63  scales(&d);
64  if (weight_variable>=0)
65  { if (!scale_ok(&d,weight_variable,RATIO_SCALE))
66     { sprintf(sbuf, "\nWeight variable %.8s must have ratio scale!",
67       d.varname[weight_variable]); sur_print(sbuf);
68       WAIT; if (scale_check==SCALE_INTERRUPT) return(-1);
69     }
70     scale_error=0;
71   }
72   for (i=0; i<d.m_act; ++i)
73   { if (!scale_ok(&d,d.v[i],SCORE_SCALE))
74     { if (!scale_error)
75       sur_print("\nInvalid scale in variables: ");
76       scale_error=1;
77       sprintf(sbuf,"%.8s ",d.varname[d.v[i]]); sur_print(sbuf);
78     }
79   }
80   if (scale_error)
81   { sur_print("\nIn MEAN score scale at least is expected!");
82     WAIT; if (scale_check==SCALE_INTERRUPT) return(-1);
83   }
84 return(1);
85 }
```

The task of this function is to check the scale types of variables selected for the analysis. In small data sets written in the edit field, the scale types of the variables (columns) cannot be given and then no checks are performed; test_scaletypes will simply return 1 which means that everything is OK. However, in data sets saved in SURVO 84C data files, each variable can be labelled with a one character label (mask column #3) which tells the scale type. For example, variables with a ratio scale are labelled with ‘R’ (discrete) or with ‘r’ (continuous) or with ‘F’ (variable is...
a frequency). If the user omits these labels (each scale label is then ' '), SURVO 84C will skip all scale checks.

In any case, at first the scales function is called to remove variables which have the scale type label ' - ', which means that the variable in question has no scale at all. For example, 'names' and 'addresses' are typically variables (fields) without a scale. Of course, a careful user does not select such variables for computations, but it is safer to have an extra check by the scales function in order to avoid harmful consequences.

On lines 66-74 the program tests the scale of the weight variable (if it is used). It is done by using the scale_ok function which is set to require RATIO_SCALE for the weight variable. RATIO_SCALE is a predefined (in survodat.h) string constant " RrF" telling the permitted scale type alternatives.

If the scale is not OK, an error message is displayed (on lines 70-71). The continuation depends on the value of the SURVO 84C system parameter scale_check. This parameter can be set to 0, 1 or 2 by the user where 0 means that scale_ok always returns 1 and no warning error messages are given, i.e. everything is accepted. The value scale_check=1 implies that messages are given as warnings, but the analysis can be continued. At the strictest level (value SCALE_INTERRUPT=2) the process is actually interrupted as we can see on line 72.

The remaining lines of test_scaletypes are devoted to corresponding checks for active variables which now should have a SCORE_SCALE at least. See how the d.v[] array selects the d.m_act variables from all d.m variables. (In our example d.m=5, d.m_act=3 and d.v[0]=2, d.v[1]=3, d.v[2]=4.)

The error messages and warnings are given by producing an output string by the standard sprintf function (usually to a global buffer sbuf of max. 256 characters) and then yielding the output by sur_print(sbuf).

The next function to be introduced is space_allocation:

```c
94 space_allocation()
95 {
96     sum=(double *)malloc(d.m_act*sizeof(double));
97     if (sum==NULL) { not_enough_memory(); return(-1); }
98     f=(long *)malloc(d.m_act*sizeof(long));
99     if (f==NULL) { not_enough_memory(); return(-1); }
100    w=(double *)malloc(d.m_act*sizeof(double));
101    if (w==NULL) { not_enough_memory(); return(-1); }
102    return(1);
103 }
104
105 not_enough_memory()
106 {
107     sur_print("\nNot enough memory! (MEAN)\n");
108     WAIT;
109 }
```
This function allocates memory for arrays \texttt{sum}, \texttt{f} and \texttt{w}, which all should have \texttt{d.m\_act} elements.

It is strongly recommended to use dynamic memory allocation for all working space which is dependent on the size of the data set. Then no theoretical limits appear for the number of variables, etc. In practice there are always some limits. On the 16 bit micros we typically have still the 64KB limit for a single array unless the huge memory model is used.

Since errors in memory allocation may have very surprising consequences, it is, of course, possible to start with fixed dimensions and later when all the space requirements are clear, dynamic arrays are established.

For example, the lines 13-16 in the main function could read:

```c
#define MAX 100
double sum[MAX];       /* sums of active variables */
long f[MAX];           /* frequencies */
double w[MAX];         /* sums of weights */
```

and \texttt{space\_allocation} is not needed at all, but this should be a temporary arrangement only.

The data set will be scanned by the \texttt{compute\_sums} function:

```c
compute_sums()
{
    int i;
    long l;
    n=0L;
    for (i=0; i<d.m\_act; ++i)
    {
        f[i]=0L; w[i]=0.0; sum[i]=0.0; 
    }
    sur_print("\n");
    for (l=d.l1; l<=d.l2; ++l)
    {
        double weight;
        if (unsuitable(&d,l)) continue;
        if (weight_variable==-1) weight=1.0;
        else
        {
            data_load(&d,l,weight_variable,&weight);
            if (weight==MISSING8) continue;
        }
        ++n;
        sprintf(sbuf,"%ld ",l); sur_print(sbuf);
        for (i=0; i<d.m\_act; ++i)
        {
            double x;
            if (d.v[i]==weight_variable) continue;
            data_load(&d,1,d.v[i],&x);
            if (x==MISSING8) continue;
            ++f[i]; w[i]+=weight; sum[i]+=weight*x;
        }
    }
}
```

At first, the work space is cleared (lines 116-118) and then the rest of the function consists of a loop for active observations (from \texttt{d.l1} to \texttt{d.l2}).
In this loop the function unsuitable checks (line 125) whether the conditions (set by conditions in the main module) are met in the current observation j. If not, the rest of the loop is skipped.

If the observation is accepted, first the value of the possible weight variable is read by the data_load function (line 129). If weight is missing (line 130), the rest of the loop is skipped. If there is no weight variable, weight=1.0 is selected (line 126).

Thereafter the number of cases n is increased by one and the order of the current observation is displayed on the screen to indicate that the run is going on (lines 132-133).

In the inner loop (lines 134-142) all the active variables are scanned and the cumulative sums updated. However, the weight variable is skipped (on line 138). Similarly, possible missing values of active variables are omitted. By comparing n to f[i] we can see the number of missing observations in each variable separately.

The final task of the !MEAN module is to give the results by calling the printout function:

```c
printout()
{
    int i;
    char line[LLENGTH];
    char mean[32];

    output_open(eout);
    sprintf(line," Means of variables in %s N=%ld%c",
           word[1],n,EOS);
    if (weight_variable>=0)
    {
        strcat(line," Weight=");
        strncat(line,d.varname[weight_variable],8);
    }
    print_line(line);
    strcpy(line," Variable     Mean     N(missing)");
    print_line(line);
    for (i=0; i<d.m_act; ++i)
    {
        if (d.v[i]==weight_variable) continue;
        if (w[i]==0.0)
            sprintf(line," %-8.8s            -  %6ld",d.varname[d.v[i]],
                    n-f[i]);
        else
            {
                fnconv(sum[i]/w[i],accuracy+2,mean);
                sprintf(line," %-8.8s %s  %6ld",d.varname[d.v[i]],
                        mean,n-f[i]);
            }
        print_line(line);
    }
    output_close(eout);
}
```

The `printout()` function is responsible for printing the results of the !MEAN module. It opens the output file (eout), formats the title of the output, and then iterates over all active variables, printing their mean and the number of missing values. The function handles the case where the weight variable is ignored in calculations.
At first the output file/device `eout` is opened by the `output_open` function. Thereafter lines can be written to `eout` by the `output_line` function (called in the function `print_line` on line 183). The lines are appended to the file. So no previous results are overwritten.

The SURVO 84C library function `output_line` writes also lines in the current edit field provided that the third argument (here `results_line`) gives a valid line number. Remember that the first line for the results was optional in the MEAN operation and we set `results_line=0` (on line 42) if that line label was missing.

`print_line` (lines 180-185) is only an auxiliary function to keep an eye on the current output line in the edit field.

It is a practice in SURVO 84C that the numerical accuracy of the printed numbers can be controlled by the user. This happens by using the system parameter `accuracy` (typically set to the value 7 in SURVO.APU) which gives the desired number of significant digits and such. The writers of the modules must take the current value of `accuracy` into account when selecting the printout parameters. The library function `fnconv` is often useful in this task. Here (on line 171) it formats the means. `accuracy+2` gives the total length of the resulting string `mean`; we must have one extra place for sign and one for the decimal point.

These 185 lines constitute the whole `!MEAN` module in its source form. Since several library functions were employed and there are many ‘hidden’ or optional properties included, the total amount of code after compiling and linking is about 60KB. However, if the module grows, the actual code size is not growing proportionally. For example, `!MEAN` can be considered a tiny special case of the `!CORR` module which computes standard deviations and correlations in addition to means, but the size of `!CORR` is only 6KB more than the size of `!MEAN`. Thus it is profitable to create modules with several tasks and options.

All SURVO 84C compilands of SURVO 84C modules have to be compiled in the large memory model because the SURVO 84C libraries (`SURVO.LIB`, `SURVOMAT.LIB`, etc.) are available in this model only. Thus, the `!MEAN.C` file is compiled by the command

```
CL /c /AL !MEAN.C
```

and it is linked by

```
LINK !MEAN,,NUL.MAP,SURVO /STACK:4000 /NOE.
```

`!MEAN` was made and presented only for illustration. Source codes for selected true SURVO 84C modules are available separately.
Each module (as an .EXE file) is normally saved in the SURVO 84C system directory (typically C:\E:) and activated by the user as MEAN. During the testing stage, it can be activated from any disk or path. For example, if !MEAN.EXE is on the disk A:,

A!:MEAN DATA1,11

is a valid command in SURVO 84C.

4. Edit field

One important link between the main program of SURVO 84C and its modules is the edit field. It materializes our idea of the editorial approach.

Most of the modules read something from the edit field and write results in it. This is done by using certain global variables and library functions.

After the s_init call we have the access to the edit field through the following global variables:

```c
char *z;       /* pointer to edit field */
int ed1,ed2;   /* length of edit line + control column */
int edshad;    /* max. # of shadow lines in edit field */
int *zs;       /* indices of shadow lines */
int r,r1,r2,r3; /* current line on the screen */
int c,c1,c2,c3; /* current column on the screen */
```

The edit field is simply a sequence of $ed1\times ed2$ characters starting from a character pointed to by $z$. Thus the $j^{th}$ line in the edit field consists of characters $*(z+(j-1)*ed1+i)$, $i=0,1,\ldots,ed1-1$, where the first one, $*(z+(j-1)*ed1)$, is the control character.

Use of direct references through $z$ should, however, be avoided, since we do not guarantee that this setup will be valid in future implementations. Therefore we recommend that the library functions edread and edwrite should always be employed in reading and writing.

Their current listings could be the following:

```c
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include "survo.h"

extern char *z;
extern int ed1,ed2;

edread(x,lin)
char x[];       /* result as a null terminated string */
int lin;        /* line number */
{
  strncpy(x,z+(lin-1)*ed1,ed1);
  x[ed1]=EOS;
```
The window on the screen (i.e. the visible part of the edit field) is
maintained by the variables $r, r1, r3, c, c1, c3$.

The current size of the window is $r3$ lines and $c3$ columns (plus the
control column). In that window the location of the cursor is $(r, c)$, the first
visible edit line is $r1$ and the first column is $c1$. Hence the current position
of the cursor in the edit field is line=$r1+r-1$ and column=$c1+c-1$.

For example, the character indicated by the cursor can be read as follows:

```
    char ch;
    char x[LLENGTH];
    edread(x,r1+r-1);
    ch=x[c1+c-1];
```

The module can change the position of the cursor and even
the position of the window by updating variables $r, c, r1, c1, c3$. In that case the $s$ _end
function must be called once before the return to the main program.

For example, the following `SEEK` module finds the first edit line
starting with a selected word and places the cursor to the first position on that
line. When necessary, the window is moved. If the word is not found, an
error message is displayed and the original display restored.

```
1 /* !seek.c 28.3.1986/SM (28.3.1986)
2 **   SEEK <word>
3 */
4
5 #include "survo.h"
6 #include "survoext.h"
7
8 main(argc,argv)
9     {  
10     int argc; char *argv[1];
11     {  
12     sur_print("\nUsage: SEEK <word>\n");
13     WAIT; return;
14     }  
15     if (argc==1) return;
16     s_init(argv[1]);
17     if (g<2)  
18     {  
19     sur_print("\nUsage: SEEK <word>\n");
20     WAIT; return;
21     }  
22     for (j=1; j<r2; ++j)  
23     {  
24     
```

All the edit lines are scanned (until success) by the loop starting from line 23. The current line is read as string \textit{x} (line 25) and the actual line (\textit{x+1}) without the control character is divided into words by the library function \textit{split} (line 26). Here only the first word (\textit{w[0]}) is of interest.

On line 27 \texttt{strcmp} compares \textit{w[0]} with \textit{word[1]} (the word given by the user). If they are the same, a proper window for displaying the line is selected (29-36) and the module ends by updating the parameters by the \texttt{s_end} call. If the words are not the same, the search continues and in an entirely unsuccessful case an error message is displayed (on lines 41-43).

5. Shadow lines

Various display effects (color, underlining, reversed video, etc.) and other attributes related to characters and edit lines are maintained by shadow lines. Normally an edit line has no shadow line, but when at least one character is typed in special display mode (turned on by the \texttt{FORMAT} key, for example), the SURVO 84C system creates a shadow line for the current line. Shadow lines are as long as normal edit lines, i.e. \texttt{ed1} bytes and they are saved in the order they are created after the last normal edit line (\texttt{ed2}).

The shadow lines may contain any kind of characters. Space (blank) is the default and means normal display on the screen. Characters ‘1’, ‘2’, ..., ‘7’ are reserved for the current palette of colors (or display effects). Their actual meaning can be controlled by the user (by editing the auxiliary file SURVO.APU). These and other control codes are also used in printing to produce various special effects.

The total amount of shadow lines is limited by the system parameter \texttt{edshad} (default is 20). This limit may, however, be changed by the RE-DIM operation. If the user tries to exceed the current limit, the system gives a warning.

If a shadow line becomes empty, the system frees it for subsequent use.
in the same edit field.

The shadow lines can be read and written as normal edit lines. The index of the shadow line for the \( j^{th} \) edit line is \( z_s[j] \). If there is no shadow line, \( z_s[j] = 0 \).

The library function \texttt{shadow\_create} is used to create new shadow lines and \texttt{shadow\_test} frees the shadow line if it consists of spaces only.

Normally the modules have no need to use shadow lines.

To illustrate working with shadow lines, we have made a small module \texttt{!SHADOW} which creates and fills all the shadow lines of specific edit lines with a selected character. For example, \texttt{SHADOW 6,10,7} turns all characters on lines 6-10 to inverse mode and \texttt{SHADOW 1,END} frees each shadow line in the current edit field.

When referring to edit lines, both line numbers and line labels may be used in SURVO 84C. Line labels are one character symbols written in the control column of the edit field. Thus in modules which take line labels as their parameters (as \texttt{!SHADOW} above) both alternatives must be supported. This is done simply by using the library function \texttt{edline2}. It was employed twice in \texttt{!SHADOW} (lines 22-23).
6. Space allocation

SURVO 84C modules are usually large model programs and compiled with the /AL option of the Microsoft C compiler. Small modules could also be in small model mode, but the current SURVO 84C libraries support only the large model. In large model programs there are no limitations for the size of code and data except the total memory available. In 16 bit micros we additionally have the limit 64KB for each data item (array). We have the same limitation, too, for the code in each compiland, but this is never a real problem, since one module (if reasonably written) is divided into compilands of much smaller size.

Within these limitations each module should be written so that space is allocated according to each application separately. This means that all vectors and matrices, etc. should get their dimensions dynamically during the run.

However, temporary arrays whose sizes depend on the line length of the edit field, are typically dimensioned by using the SURVO 84C system constant \texttt{LLLENGTH} and its multiplicities. The current value of \texttt{LLLENGTH} is 256 and it implies the maximum line length of the edit field to be 253. Another constant is \texttt{LLNAME} (current value 64) which is used for names of files (pathnames) etc.

In some cases the maximum number of columns (or maximum number of words or numbers on a single edit line) is critical for some arrays. The system constant \texttt{EEP4} (current value 100) gives that limit. The counterpart of \texttt{EEP4} in the SURVO.APU file is the system parameter \texttt{eep4} which may be used in dynamic space allocation for arrays related to number of columns in the edit field.

Matrices should always be defined as one-dimensional arrays and their elements should be saved columnwise. Thus the element on row \(i\) and column \(j\) of a \(m \times n\) matrix \(A\) will be \(A[i+n*j]\) where \(i=0,1,...,m-1\) and \(j=0,1,...,n-1\). In all arrays the base value for the indices is 0. In output, however, the base value is always 1.

Since double precision should normally be used in matrix computations, the largest square matrix (within the 64KB limit) is \(90 \times 90\).

Some of the library functions make their own space reservations. For example, when a data set (\texttt{SURVO_DATA}) is opened by the \texttt{data_open} function, memory is allocated for all arrays pointed to by members of this structure.
7. Include files

Various system constants, variables and macros are defined in include files `survo.h`, `survoext.h` and `survodat.h`.

In all standard SURVO 84C compilands, `survo.h` and `survoext.h` must be included. In compilands working with SURVO 84C data (lists, tables, files, etc.) or using extra specifications, also `survodat.h` should be observed.

The most important constants defined in `survo.h` are the following (current values in parentheses):

- **EEP4** - Maximum number of ‘words’ on one edit line (100)
- **LENGTH** - Maximum length of a ‘word’ or a ‘line’ (256)
- **LNAME** - Maximum length of a pathname (64)
- **MAXP** - Maximum number of ‘words’ in a command (24)

The following macros, defined in `survo.h`, are available for the screen control etc.:

- **WAIT** - halts the process and displays the message ‘Press any key!’ until the user has pressed a key.
- **CLS** - clears the screen.
- **LOCATE((int)r,(int)c)** - The cursor will be located on line `r` and column `c`.
- **ERASE** - erases the current line.
- **BEEP** - gives a sound signal.
- **CURSOR_OFF** - makes the cursor invisible.
- **CURSOR_ON** - displays a normal cursor.
- **CURSOR_INS** - displays an extended cursor (used in insert mode).
- **CURSOR_POS((int *)prow,(int *)pcol)** - saves the current row and column of the cursor in `*prow` and `*pcol`.
- **SCROLL_UP((int)lin1,(int)lin2,(int)n)** - scrolls the lines from `lin1` to `lin2` `n` steps upwards.
- **SCROLL_DOWN((int)lin1,(int)lin2,(int)n)** - scrolls the lines from `lin1` to `lin2` `n` steps downwards.
- **PR_UP** - moves the cursor one step upwards.
- **PR_DOWN** - moves the cursor one step downwards.
- **PR_RIGHT** - moves the cursor one step to the right.
Programming SURVO 84 in C

8. Libraries

The ready-made tools for programming SURVO 84C modules have been collected in the following libraries:

- **SURVO.LIB**: Functions for general system control, management of the edit field, data management, specifications, sucros, and prompts
- **SURVOMAT.LIB**: Routines for matrix management and algebra
- **DISTRIB.LIB**: Continuous statistical distributions (by T. Patovaara)

All the functions in these libraries have been compiled in the large memory model. The functions will now be described separately for each library.
8.1 Library SURVO.LIB

activated

Summary
int activated(data, character)
SURVO_DATA *data; /* pointer to data structure */
char character; /* activation character */

Description
The activated function finds the first variable which has been activated
by character in data opened by data_open or data_open2.

Return Value
activated returns # of variable or -1 if no variable has been activated by
character.

See Also
mask, varfind

Example
int i;
int weight_variable;
SURVO_DATA dat;

i=data_open("TEST", &dat);
if (i<0) return;
weight_variable=activated(&dat,'W');

  **  **

conditions

Summary
int conditions(data)
SURVO_DATA *data; /* pointer to data structure */

Description
The conditions function reads and tests the IND and CASES specifica-
tions according to `data` opened by `data_open` or `data_open2`.

`conditions` can be called only once in each SURVO 84C module and it forms the basis for data scanning where `unsuitable` is the function for eliminating those observations (records) which do not satisfy the IND and CASES restrictions.

**Return Value**

`conditions` returns 1 if IND and CASES specifications have been written correctly. In case of an error -1 is returned.

**See Also**

`unsuitable`

**Example**

```c
int i;
SURVO_DATA dat;

i=data_open("TEST", &dat);
if (i<0) return;
i=conditions(&dat);
if (i<0) { data_close(&dat); return; }

***
```

### create_newvar

**Summary**

```c
int create_newvar(data, name, type)
SURVO_DATA *data; /* pointer to data structure */
char *name;       /* name of new variable */
char type;        /* type 1,2,4,8 or S of new var. */
int len;          /* length of field (S type only) */
```

**Description**

The `create_newvar` function creates a new variable with the name `name` and of the type `type` for data `data` which has to be opened by `data_open2` of the form `data_open2(name, data, 1, 0, 0)`.

The length of the field in case of a string (S) variable, is given by `len`. In numeric variables the length is determined by `type` and `len` is not used.
Return Value

\texttt{data\_load} returns the index \((0, 1, 2, \ldots, data->m-1)\) of the new variable and -1 if there is no room for new variables or the data representation does not permit creation of new variables.

See Also

\texttt{data\_save}

Example

```c
int i;
long j;
SURVO\_DATA dat;

i=data\_open2("TEST",\&dat,1,0,0);
if (i<0) return;
i=create\_newvar(\&dat,"COUNT","2");
if (i>=0)
  for (j=1L; j<=dat.n; ++j)
    data\_save(\&dat,j,3,MISSING8);

/* open TEST, create a new variable COUNT of integer type and save missing values in it. */

***

\texttt{data\_alpha\_load}

Summary

```c
int data\_alpha\_load(data,j,i,string)
SURVO\_DATA *data; /* pointer to data structure */
long j; /* # of observation (record) */
int i; /* # of variable (field) */
char *string; /* pointer to data value */
``` 

Description

The \texttt{data\_alpha\_load} function reads the value of the \(j^{th}\) observation in variable \(i\) \((i=0,1,\ldots, data->m-1)\) as a null-terminated string from \texttt{data} opened by \texttt{data\_open} or \texttt{data\_open2}.

Only variables of string (S) type can be loaded by \texttt{data\_alpha\_load}.
Return Value

data_alpha_load returns 1 if the value is found. Otherwise -1 is returned.

See Also

data_load

Example

```c
int i;
long j;
char value[LLENGTH];
SURVO_DATA dat;

i=data_open("TEST", &dat);
if (i<0) return;
sprintf(sbuf, "\nValues of variable %s:\", dat.varname[3]);
sur_print(sbuf);
for (j=dat.l1; j<=dat.l2; ++j) {
    data_alpha_load(&dat, j, 3, value);
    sprintf(sbuf, "\nValue of var. # %d in obs. # %ld is %s",
            i+1, j, value);
    sur_print(sbuf);
}
/* open TEST and print values of var. #4 as strings */
```

---

**data_close**

Summary

```c
int data_close(data)
SURVO_DATA *data; /* pointer to data structure */
```

Description

The `data_close` function closes `data` opened by `data_open` or `data_open2` and frees the space allocated for `data`.

Return Value

There is no return value.
See Also
  data_open, data_open2

Example
SURVO_DATA dat;
data_close(&dat);

***

data_load

Summary
int data_load(data,j,i,px)
SURVO_DATA *data; /* pointer to data structure */
long j;           /* # of observation (record) */
int i;            /* # of variable (field) */
double *px;       /* pointer to data value */

Description
  The data_load function reads the value *px of the jth observation in
  variable # i (i=0,1,...,data->m-1) from data opened by data_open
  or data_open2.
  Both numeric (N) and string (S) fields can be loaded by data_load. In
  the latter case the string value is converted to double by the standard function
  atof.

Return Value
  data_load returns 1 if the value is found. Otherwise -1 is returned.

See Also
  data_alpha_load

SURVO.LIB
Example
int i;
long j;
double x;
SURVO_DATA dat;

i=data_open("TEST", &dat);
if (i<0) return;
sprintf(sbuf, "\nValues of variable %s:", dat.varname[3]);
sur_print(sbuf);
for (j=dat.l1; j<=dat.l2; ++j)
{
  data_load(&dat, j, 3, &x);
  sprintf(sbuf, "\nValue of var. # %d in obs. # %ld is %f", i+1, j, x); sur_print(sbuf);
}
/* open TEST and print values of var. #4 */

***

data_open

Summary
int data_open(name, data)
char *name;       /* name of SURVO 84C data */
SURVO_DATA *data; /* pointer to data structure */

Description
The data_open function opens the SURVO 84C data specified by name. 
data is a pointer to the SURVO_DATA structure. The structure type 
SURVO_DATA is defined in survodat.h as follows:

#define SURVO_DATA struct survodata

SURVO_DATA
{
  SURVO_DATA_MATRIX d1; /* data matrix structure */
  SURVO_DATA_FILE d2;   /* data file structure */
  int type;             /* 1=data matrix 2=data file */
  /* 3=data list 4=matrix file */
  char *pspace;         /* pointer to allocated space */
  int m;                /* # of variables */
  long n;               /* # of observations */
  int m_act;            /* # of active variables */
  long l1, l2;          /* selected observations */
}
int typelen; /* # of attributes for variables */
int *v; /* indices of active variables */
char **varname; /* names of variables */
int *varlen; /* lengths of variables */
char **vartype; /* types etc. of variables */
int *varpos; /* positions of variables */
}

SURVO 84C supports four different forms of data. Data may be located
in the current edit field as a data matrix or a data list, or in a SURVO 84C
data file or in a SURVO 84C matrix file. All forms of data can be accessed
by data_open.

Usually, when writing SURVO 84C modules it is not necessary to
know the type of the data (given by the structure member type).

data_open(name, data);

is equivalent to

data_open2(name, data, 0, 0, 0);

which means that, in case of a data file, the data is opened
with space allocated for defined variables (fields) only,
with short names (of 8 bytes) for variables,
and without text information.
This is usually sufficient in statistical operations.

Return Value

data_open returns 1 if the file was successfully opened and -1 otherwise.
In the latter case an error message SURVO 84 data 'name' not found!
is displayed.

See Also

data_open2, data_close, data_load, data_alpha_load

Example

int i;
SURVO_DATA dat;
i=data_open("TEST", &dat);
if (i<0) return;

***
data_open2

Summary

int data_open2(name, data, p1, p2, p3)
char *name;       /* name of SURVO 84C DATA */
SURVO_DATA *data; /* pointer to data structure */
int p1;           /* space for variables indicator */
int p2;           /* name length indicator */
int p3;           /* text indicator */

Description

The data_open2 function opens the SURVO 84C data specified by name.
data is a pointer to the SURVO_DATA structure (See data_open).
The parameters p1,p2,p3 indicate various extensions when data is a
SURVO 84C data file.
If p1=0, space will be allocated for defined variables (fields)
only (i.e. for data->m variables).
Otherwise space is available for all possible variables
(i.e. for data->m1 variables).
If p2=0, space will be allocated for short names (of 8 bytes)
of variables (fields),
otherwise space is available for full length
(data->d2.l bytes) names of variables.
If p3=0, no text information is loaded.
Otherwise space is allocated for general text information
saved in the data file.
The text lines are referred to by pointers
data->d2.fitext[i], i=0,1,...,data->d2.textn.

Return Value

data_open2 returns 1 if the file was succesfully opened and -1 other-
wise. In the latter case the error message SURVO 84 data 'name' not found!
is displayed.

See Also

data_open, data_close, data_load, data_alpha_load
Example

```c
int i;
SURVO_DATA dat;
i=data_open2("TEST",dat,1,1,1);
if (i<0) return;

***
```

**data_save**

**Summary**

```c
int data_save(data,j,i,x)
SURVO_DATA *data; /* pointer to 84 data structure */
long j;       /* # of observation (record) */
int i;        /* # of variable (field) */
double x;     /* value to be saved */
```

**Description**

The `data_save` function saves the value `x` of the `j`th observation in variable `# i` (`i=0,1,...,data->m-1`) for `data` opened by `data_open` or `data_open2`.

Only numeric values can be saved by `data_save`. If the field for saving is a string field, value `x` is converted to a string.

**Return Value**

`data_save` returns -1 if the field for saving is protected or the data representation does not permit saving.

**See Also**

`data_load`, `create_newvar`
Example

```c
int i;
long j;
SURVO_DATA dat;

i=data_open("TEST",&dat);
if (i<0) return;
for (j=1L; j<=dat.n; ++j)
    data_save(&dat,j,3,MISSING8);

/* open TEST and save missing values in
 all observations of field #3 */
```

***

edline2

Summary

```c
int edline2(label,j,error)
char *label;       /* null-terminated string */
int j;             /* first edit line to be scanned */
int error;         /* display of error */
```

Description

The edline2 function searches for the first occurrence of label in the control column starting from line j in the edit field. If the edit line is not found, message Line 'label' not found! is displayed. However, if error is 0, no message is produced.

label can be a line number 1,2,... or a line label consisting of one character or of the form END, END-1, END-2, END+1, etc., where END refers to the last non-empty line in the current edit field or of the form CUR, CUR+1, etc., where CUR refers to the current line. Thus edline2 covers all the possibilities the user may employ when referring to lines in SURVO 84C operations.

Return Value

edline2 returns the index of the first edit line found with label and 0, if no edit line with label exists in the current edit field.
See Also
lastline2

Example
Assume that we have in the edit field:

```
1 *    1 *
2 *    2 *
3 *    3 *
4 A This line should be found!    4 A This line should be found!
5 *
```

Then

```c
unsigned int j;
j = edline2("A", 1, 1);
returns j=4.
```

* * *

edread

Summary

```c
int edread(x, j)
char *x;       /* storage location for input string */
unsigned int j;              /* number of edit line */
```

Description

The function `edread` reads line `j` from the current edit field to `x` as a null-terminated string. `x[0]` will be the control character of the edit line and the length of `x` is `edl`. Thus the terminating spaces are also in `x`.

Space for `x` must be allocated before the `edread` call; it should be at least `LLLENGTH` characters.

Return Value

There is no return value.

See Also
edwrite
Example
Assume that we have in the edit field:

```
  7 *
  8 *PRINT 11,20
  9 *
```

Then

```c
char x[LLENGTH];
edread(x,8);
gives x="*PRINT 11,20                           
where strlen(x)=ed1 (width of the edit field + 1).
```

** **

edwrite

Summary

```c
int edwrite(x,j,pos)
char *x;           /* null-terminated string */
unsigned int j;    /* number of edit line */
int pos;           /* first position on edit line */
```

Description

Function edwrite writes the string x on the line j in the current edit field from the column pos onwards. If x is longer than the edit line length permits, the extra characters are not written.

Return Value

There is no return value.

See Also

edread, output_line

Example

Assume that we have in the edit field:

```
  7 *
  8 *Result: _
  9 *
```
Then

char x[]="123.456"
edwrite(x,8,9);

gives

7 *
8 *Result: 123.456
9 *

Applications

edwrite is the standard tool in writing results of edit operations in the edit field. In operations producing larger output both in the edit field and in output files, \texttt{wline} is to be used instead of \texttt{edwrite}.

\* \* \*

\texttt{empty\_line}

Summary

\begin{verbatim}
int empty_line(s,len)
char *s;           /* string */
int len;           /* length of string */
\end{verbatim}

Description

The \texttt{empty\_line} function tests whether the string \texttt{s} consists (for the \texttt{len} first bytes) of spaces (blanks) only.

Return Value

\texttt{empty\_line} returns 1 if the entire string \texttt{s} or its \texttt{len} first bytes are spaces and otherwise 0.

Example

\begin{verbatim}
char x[LENGTH];
edread(x,r1+r);
i=empty_line(x+1,c2);
/* i=1, if the line after the activated line is empty
** and i=0, if it is not empty.
*/
\end{verbatim}

\* \* \*

\texttt{SURVO.LIB}
fconv

Summary

int fconv(number, format, string)
double number;       /* number to be converted */
char *format;        /* format to be used */
char *string;        /* string result */

Description

The fconv function converts the digits of the given number to a null-terminated character string and stores the result in string.

The conversion takes place according to the given format which is a null-terminated string of form "1234.123" or "%8.3f".

The format "" means the shortest possible representation of number as string.

Return Value

fconv returns 1 if the number is successfully converted and -1 if the format is too restrictive.

See Also

fnconv

Example

double x=3.14159265;
char format[]="123.123";
char result[32];
fconv(x, format, result);  /* result="  3.142" */

* * *
**fi_create**

**Summary**

fi_create(name, len, m1, m, n, f, extra, textn, textlen, text, varname, varlen, vartype)

- char *name;    /* name of data file */
- int len;       /* record length */
- int m1;        /* max # of fields */
- int m;         /* # of fields */
- long n;        /* # of observations */
- int f;         /* max field name length */
- int extra;     /* field attribute length */
- int textn;     /* # of comment lines */
- int textlen;   /* length of comment line */
- char *text[];  /* pointers to comment lines */
- char *varname[]; /* names of fields */
- int varlen[];  /* field lengths */
- char *vartype[]; /* field attributes */

**Description**

The **fi_create** function creates a new SURVO 84C data file with a pathname `name`. If the path is not given, the current data pathname given by the global variable `edisk` is used. The default extension is `.SVO`.

If `n>0`, `n` missing observations will be saved.

The data file has the following structure:

<table>
<thead>
<tr>
<th>Header fields:</th>
<th>64 bytes (46 bytes in use)</th>
<th>bytes</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;SURVO 84C DATA &quot;</td>
<td>char</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>record length (len)</td>
<td>int</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>max # of fields (m1)</td>
<td>int</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td># of fields (m)</td>
<td>int</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td># of observations (n)</td>
<td>long</td>
<td>4</td>
<td>22</td>
</tr>
<tr>
<td>max length of field name (f)</td>
<td>int</td>
<td>2</td>
<td>26</td>
</tr>
<tr>
<td>length of field attr. (extra)</td>
<td>int</td>
<td>2</td>
<td>28</td>
</tr>
<tr>
<td># of comment lines (textn)</td>
<td>int</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>length of comment line (textlen)</td>
<td>int</td>
<td>2</td>
<td>32</td>
</tr>
</tbody>
</table>
start of comments (text) long 4 34
start of field descr. (var) long 4 38
start of data (data) long 4 42

Comments: text=64
Length of comments textn*textlen bytes

Field descriptions: var=text+textn*textlen
Following information repeated m1 times (f+extra bytes for each field):
  position in record (varpos) int 2 0
  length of field (varlen) int 2 2
  type (1,2,4,8,S) (vartype) char 1 4
  activation char 1 5
  protection char 1 6
  other mask bytes char extra-7 7
  name (varname) char f extra

Data: data=var+m1*(f+extra)
       =64+textn*textlen+m1*(f+extra)
Observation j starts from data+(j-1)*len

Return Value
fi_create returns 1 if the file is successfully created and -1 otherwise.
If a data file with the same name already exists, fi_create asks for permission to overwrite.

Application
In practice, SURVO 84C data files are created automatically by various FILE operations. They use fi_create as a subroutine. A direct fi_create call is seldom needed.

* * *

fnconv

Summary
int fnconv(number,length,string)
double number; /* number to be converted */
int length; /* length of the result */
char *string; /* string result */
Description
The `fnconv` function converts the digits of the given `number` to a null-terminated character string and stores the result in `string`. The format of the result is selected so that the length of `string` will be `length`. Exceptionally large numbers are converted to a floating point (exponent) form and the length of `string` may then exceed `length`.

Return Value
There is no return value.

See Also
fconv

Example
double x=3.14159265;
char result[32];
fnconv(x,7,result); /* result=" 3.1416" */

***

`hae_apu`

Summary
int hae_apu(s,t)
char *s;       /* keyword in SURVO.APU */
char *t;       /* value of keyword as a string */

Description
The `hae_apu` function searches for the keyword `s` in the system file `SURVO.APU` which contains the values of the SURVO 84C system parameters. The value of the keyword is copied as a null-terminated string to `t`.

Return Value
`hae_apu` returns 1 if the keyword `s` is found and 0 otherwise.

See Also
s_init

---

SURVO.LIB
Example
char value[16];
int ep4;
ep4=EP4;
if (hae_apu("ep4",value)) ep4=atoi(value);

replaces the default value EP4 of system parameter ep4 by a value found in SURVO.APU in the form ep4=value.

Applications
hae_apu does not read the file SURVO.APU itself, but rather a buffer which has been created when SURVO 84C is initialized or the SETUP command has been activated.

hae_apu is not needed very often, since most of the system parameters maintained by SURVO.APU have been read by the SURVO 84C main module and appear as global variables in any SURVO 84C module after the s_init call (See s_init).

***

init_remarks, rem_pr, wait_remarks

Summary
int init_remarks()

int rem_pr(string)
char *string;  /* output line */

int wait_remarks(type)
int type;     /* type of prompt */

Description
These functions are to be used in supplementary SURVO 84C modules not reported in the inquiry system. Substantially, these functions emulate the behaviour of the inquiry system of SURVO 84C. They are called when the user has activated the operation with insufficient parameters.

The init_remarks function initializes a temporary window for remarks to be printed on consecutive lines, possibly on several pages.

The rem_pr function prints one line of remarks given as string.

The wait_remarks function halts the display temporarily and gives two kinds of prompts according to type.
If type=1, the prompt is
   Next page by 'space'  |   Load lines by '+'  |   Interrupt by ENTER!

If type=2, the prompt is
   Load lines by '+'  |   Interrupt by ENTER!

Thus wait_remarks(1); should be given once between pages and wait_remarks(2); after the last page.

Return Value
   There is no return value.

* * *

lastline2

Summary
   int lastline2()

Description
   Function lastline2 finds the last non-empty line in the current edit field.

Return Value
   lastline2 returns the line number. There is no error return.

See Also
   edline2

* * *

mask

Summary
   int mask(data)
   SURVO_DATA *data; /* pointer to data structure */

Description
   The mask function reads the VAR (or VARS) specification written in
   the edit field or, if it does not exist, mask reads the MASK specification
   and activates variables (fields) in data opened by data_open or data_
   SURVO.LIB
open2.
The effect of mask is only temporary. There is no change in the activation status of files. Thus if data is reopened, activation due to mask is no longer valid.

Return Value
mask returns -1 if the VARS or MASK specification is invalid. Otherwise mask returns 1.

See Also
scales, activated, varfind

Example
int i;
SURVO_DATA dat;

data_open("TEST", &dat);
if (i<0) return;

** *

cell_format

Summary
int cell_format(format, accuracy, A, m, n)
char *format;    /* format of type ###.#### */
int accuracy;    /* # of characters in one element */
double *A;       /* matrix */
int m, n;        /* # of rows and columns */

Description
The cell_format function finds a common suitable format as a string of the form ###.#### for the elements of an m*n matrix A. The length of format is given by accuracy.
cell_format usually precedes matrix_print when the suitable format is unknown for the matrix to be written.
Return Value

matrix_format always returns 1.

See Also

matrix_print, library SURVOMAT.LIB

* * *

matrix_load

Summary

int matrix_load(matr,A,rdim,cdim,rlab,clab,
    lr,lc,type,expr)
char *matr;   /* name of matrix file */
double **A;   /* pointer to matrix space */
int *rdim;    /* pointer to number of rows */
int *cdim;    /* pointer to number of columns */
char **rlab;  /* pointer to row labels space */
char **clab;  /* pointer to column labels space */
int *lr;      /* pointer to length of row label */
int *lc;      /* pointer to length of column label */
int *type;    /* pointer to type of matrix */
char *expr;   /* matrix expression (internal name) */

Description

The matrix_load function reads a matrix saved in a matrix file. It also allocates space (by malloc) for the matrix elements (of type double) and for the row and column labels. matrix_load does not allocate space for scalar parameters or for the matrix expression.

The elements of the matrix are read by columns in a one-dimensional double array pointed by *A and having the size (*rdim)*(*cdim)*sizeof(double).

Each row label has the length *lr and they are read in a one-dimensional character array pointed by *rlab and having the size (*lr)*(*rdim).

If rlab is NULL, no space is allocated and no row labels are read.

Each column label has length of *lc bytes and they are read in an one-dimensional character array pointed by *clab and having the size (*lc)*(*cdim).

If clab is NULL, no space is allocated and no row labels are read.
*type will be the type of the matrix with possible values of
*type=20  diagonal matrix,
*type=10  symmetric matrix,
*type=0   general matrix

and *expr will be the internal name of the matrix (as a matrix expression) of length 128 characters at most. Space is not allocated to *expr in matrix_load; it is the responsibility of the calling function to have 129 bytes at least for *expr.

Similarly space must be allocated for rdim,cdim,lr,lc and type before the matrix_load call.

Return Value

matrix_load returns -1 if matrix matr is not found or if space cannot be allocated for it. Upon successful completion of the function 1 is returned.

See Also

matrix_save, matrix_print, library SURVOMAT.LIB

Example

double *A;
int m,n;
char *rlab,*clab;
int lr,lc;
int type;
char expr[129];

matrix_load("MEANS",＆A,&m,&n,&rlab,&clab,
＆lr,&lc,&type,expr);

reads an m*n matrix A from a matrix file MEANS.MAT on the current data disk. The labels of rows are read in character array clab and each label has length lr. The labels of columns are read in character array rlab and each of them has length lc. In most cases lr=lc=8. The type of matrix is type and its internal name is expr.

* * *
matrix_print

Summary

```c
int matrix_print(A, m, n, rlab, clab, lr, lc,
    m2, n2, mv, nv, form, width, editline, outfile, header)
```

```c
double *A;        /* matrix */
int m, n;          /* # of rows and columns */
char *rlab, *clab; /* row and column labels */
int lr, lc;        /* lengths of row and col. labels */
int m2, n2;        /* # of selected rows/cols */
int *mv, *nv;      /* lists of selected rows/cols */
char *form;       /* format as 123.12 or %5.5g */
int width;        /* entire printing width */
int editline;     /* first edit line for the output */
char *outfile;    /* output file/device */
char *header;     /* header text */
```

Description

The `matrix_print` function writes an `m*n` matrix `A` or an `m2*n2` submatrix of it in the current edit field and/or appends the same text in a text file `outfile`.

The matrix is written in blocks of maximal width of `width` characters. Each block will be labelled with appropriate row and column labels. The first output line will be `header`.

- If `m2=m, n2=n` and `mv=nv=NULL`, the entire `A` matrix will be written.
- If `m2<=m`, `n2<=n` and `mv=nv=NULL`, the `m2` first rows and `n2` first columns of `A` will be written.
- If `mv` is not `NULL`, rows `mv[0], mv[1], ..., mv[m2-1]` (with possible values from 0 to `m-1`) are written in this order.
- If `nv` is not `NULL`, columns `nv[0], nv[1], ..., nv[m2-1]` (with possible values from 0 to `n-1`) are written in this order.

Return Value

`matrix_print` always returns 1.

See Also

- `matrix_load`, `matrix_format`, library SURVOMAT.LIB

* * *
matrix_save

Summary

int matrix_save(matr,A,m,n,rlab,clab,lr,lc,type,expr)
char *matr;   /* name of matrix file */
double *A;    /* pointer to matrix */
int m;        /* number of rows */
int n;        /* number of columns */
char *rlab;   /* row labels space */
char *clab;   /* column labels space */
int lr;       /* length of row label */
int lc;       /* length of column label */
int type;     /* type of matrix */
char *expr;   /* matrix expression (internal name) */
int nrem;     /* # of comment lines in edit field */
int remline;  /* first edit line for the comments */

Description

The matrix_save function saves matrix \( A \) and its row (rlab) and column (clab) labels in a matrix file *matr*. *matr* is a pathname with the default path given by the global variable edisk and with the default extension .MAT.

The elements of the matrix \( A \) are assumed to be in a one-dimensional double array pointed by \( A \) by columns.

Each row label has a length of \( lr \) bytes and they are in a one-dimensional character array rlab as a contiguous string.

Each column label has a length of \( lc \) bytes and they are in a one-dimensional character array pointed by clab as a contiguous string.

type is the type of the matrix with possible values of

type=20 diagonal matrix,
type=10 symmetric matrix,
type=0 general matrix,
type=-1 unknown type.

In the last case matrix_save itself determines the type.

expr is the internal name of the matrix (as a matrix expression) of a length of 128 characters at most.

Also nrem comment lines from the edit field starting from edit line remline can be saved in the matrix file. In case of no comment lines nrem=remline=0.
The matrix file has the following structure:

**Header fields:** ERC bytes (ERC=128)
"MATRIX84D m n nrem lr lc type "
appearing as an ASCII string where the first 10 bytes "MATRIX84D " are for identification and the numeric parameters

- # of rows (m)
- # of columns (n)
- # of comment lines (nrem)
- row label length (lr)
- column label length (lc)
- type of the matrix (type)

have been converted to character strings separated by blanks.

The header is followed by:

- **Comments:** nrem*ERC bytes
- **Internal name (expr):** ERC bytes
- **Column labels (*clab):** n*lc bytes
- **Rows of the matrix:** lr+8*n bytes each

If the matrix is symmetric (type=10), only the elements of the lower triangular part are saved by rows, each row preceded by its label of lr bytes.

If the matrix is diagonal (type=20), only the diagonal elements, each preceded by the row label, are saved.

The total size of the matrix file is

- m*(lr+8*n)+n*lc+(nrem+2)*ERC bytes for type=0,
- m*(lr+8*(m+1)/2)+m*lc+(nrem+2)*ERC bytes for type=10,
- m*(lr+8)+m*lc+(nrem+2)*ERC bytes for type=20.

**Return Value**

`matrix_save` returns -1 if matrix cannot be saved. Otherwise 1 will be returned.

**See Also**

- `matrix_load`, library SURVOMAT.LIB
Example
double *A;
int m,n;
char *rlab,*clab;
int lr,lc;
int type;
char expr[129];

matrix_save("MEANS",A,m,n,rlab,clab,8,8,-1,expr,0,0);

saves an $m \times n \times n$ matrix $A$ in a matrix file MEANS.MAT on the current data
disk. The labels of rows are in character array $clab$ and each label has
length 8. The labels of columns are in character array $rlab$ and each of
them has length 8. The type of matrix is -1 (unknown) and its internal
name is $expr$. No comment lines are saved from the edit field.

* * *

nextch

Summary
int nextch(display_text)
char *display_text      /* prompt text */

Description
The nextch function prompts the user to press some key by displaying
display_text on the bottom line of the screen.
nextch works also under tutorial mode (reading key strokes from the
sucro file).

Return Value
nextch returns the SURVO 84C key code of the key pressed.

See Also
tut_init, prompt
Example

```c
int m;
m=0;
while (m!=CODE_RETURN)
    m=nextch("Press ENTER!");
/* The program waits until ENTER is pressed */
```

** ** *

**output_open, output_line, output_close**

Summary

```c
int output_open(file)
char *file;      /* name of output file */

int output_line(string,file,editline)
char *string;    /* output string */
char *file;      /* name of output file */
int editline;    /* current output line in edit field */

int output_close(file)
char *file;      /* name of output file */
```

Description

The `output_open` function opens file to be used as an output file for SURVO 84C results.

The `output_line` function appends the given string to file, replacing string's terminating null character (EOS) with a newline character ('\n') in file.

Simultaneously string will be copied to editline in the current edit field, if 1<=editline<=r2. When editline overrides those limits, no error message is given, but copying is prohibited. Thus editline=0 suppresses printing in the edit field.

The `output_close` function closes file.

Return Value

`output_open` returns 1 if the file was successfully opened and -1 otherwise. There is no return value for `output_line` and `output_close`.

See Also

`edwrite`

---

SURVO.LIB
Application

The global SURVO 84C variable `eout` gives the name of the output file/device the user has selected (by OUTPUT command, for example).

Normally `eout` is opened once by

```
int i;
i=output_open(eout);
if (i<0) return;
```

Then each output line is written by

```
int ed_output_line;
    /* initialized according to situation */
char line[LLENGTH];
    /* filled with information to be written */
output_line(line,eout,ed_output_line++);
```

Finally, after all results have been written `eout` is closed by

```
output_close(eout);
```

** **

prompt

Summary

```
int prompt(question,answer,maxlength)
char *question; /* prompt text */
char *answer;   /* default/final answer */
int maxlength;  /* max length of the answer */
```

Description

The `prompt` function presents a `question` on the screen giving a default `answer` and letting the user edit it or type a new answer within the limit given by `maxlength`.

The place for the prompt can be selected by `LOCATE(row,column);`.

`prompt` works also under tutorial mode (reading the user’s answers from the sucro file).

Return Value

There is no return value. The user’s answer will be in `answer` as a null-terminated string.
See Also
tut_init, nextch

Example
cchar filename[LNAME];

strcpy(filename,"TEST");
LOCATE(r3+2,1); /* bottom line on the screen */
prompt("Name of file? ",filename,LNAME-1);

      * * *

s_end

Summary
int s_end(address)
char *address;       /* address of SURVO 84C pointers as a string */

Description
The s_end function copies the SURVO 84C system parameters which have been altered in current module back to the main module.
s_end should be called once before returning to the main program in those modules which change any of the scalar parameters r,r1,c,c1,etu,etu1,etu2,etu3,tutpos,erun,edisp.
For example, edisp may be changed from its default value 1 to to avoid redisplay of the entire screen after return.
Normally s_end is not needed at all.

See Also
s_init

Example
See also s_init.
s_end(argv[1]);

      * * *

SURVO.LIB
s_init

Summary
int s_init(address)
char *address; /* address of SURVO 84C pointers as a string */

Description
The s_init function copies the SURVO 84C system parameters from the SURVO 84C main module (parent process) to the current module (child process).

The 4-byte address of pointers is passed from the parent to the child in a form of a string address and in practice always replaced by argv[1].

After the s_init call, which should take place once in the beginning of each SURVO 84C module the following variables and parameters are available with their current values:

char *z; /* pointer to edit field */
int ed1; /* length of edit line + control column */
int ed2; /* number of lines in edit field */
int edshad; /* max. # of shadow lines in edit field */
int r; /* current line on the screen */
int r1; /* first visible edit line on the screen */
int r2; /* # number of edit lines on the screen */
int c1; /* current column on the screen */
int c2; /* first visible column on the screen */
int c3; /* # of columns on the screen */
char *edisk; /* current data disk (path) */
char *esysd; /* SURVO 84C system disk (path) */
char *eout; /* output file/device */
int etu; /* tutorial mode indicator */
char *etufile; /* current sucro file (when etu>0) */
long tutpos; /* pointer to sucro file */
int zs; /* indices of shadow lines */
int zshn; /* # of shadow lines */
int erun; /* run mode indicator (1=run mode on) */
int edisp; /* display mode after exit from current module */
char *sapu; /* buffer for SURVO.API parameters */
char *info; /* string for information between modules */
char **key_label; /* key label pointers */
char *key_lab; /* key label buffer */
char *survo_id; /* owner of the SURVO 84C copy */
char **disp_string; /* display string pointers */
int specmax; /* max # of specifications */
char *active_data; /* current SURVO 84C DATA */
int scale_check; /* scale type checking level */
int accuracy; /* accuracy for printouts */
int scroll_line; /* first scroll line for temporary displays */
int space_break; /* break indicator for space bar (1=on) */
int sdisp; /* current shadow character (display mode) */
Return Value

There is no return value.

See Also

hae_apu, s_end, tut_init

Example

A typical start of the main function in a SURVO 84C module is:

```c
main(argc,argv)
int argc; char *argv[1];
{
    if (argc==1) return;
    s_init(argv[1]);
    /* ............ */
}
```

** * * *

scale_ok

Summary

```c
int scale_ok(data,i,scale)
SURVO_DATA *data; /* pointer to data structure */
int i;            /* # of variable */
char *scale; /* list of allowed scales as a string */
```

Description

The `scale_ok` function tests whether the scale type of variable #i in data opened by `data_open` or `data_open2` belongs to the given list `scale` of scale types.

In `survodat.h` the following scale types are predefined:

```c
#define ORDINAL_SCALE       " DOoSsIiRrF"
#define SCORE_SCALE         " DSsIiRrF"
#define INTERVAL_SCALE      " DIiRrF"
#define RATIO_SCALE         " RrF"
#define DISCRETE_VARIABLE   " DNOSIRF"
#define CONTINUOUS_VARIABLE " osir"
```

where the different scale types are denoted as follows:
- no scale
  (blank)  scale unknown
D  Dichotomy (two distinct numeric values)
N  Nominal
O  Ordinal  (discrete)
o  Ordinal  (continuous)
S  Score    (discrete)
s  Score    (continuous)
I  Interval (discrete)
i  Interval (continuous)
R  Ratio    (discrete)
r  Ratio    (continuous)
F  Frequency

Return Value
scale_ok returns 1 if the scale of variable # i is found in scale. Otherwise 0 is returned. If the system parameter scale_check is 0, then scale is not checked at all and 1 is returned. However, if the scale of variable # i is '-', 0 is returned irrespective of scale and the value of scale_check.

See Also
scales

Example
int i;
int weight_variable;
SURVO_DATA dat;

i=data_open("TEST",&dat);
if (i<0) return;
weight_variable=activated(&dat,'W');
if (weight_variable>=0)
{
    if (!scale_ok(&dat,weight_variable,RATIO_SCALE))
    {
        printf("Weight variable %.8s must have ratio scale!",
               dat.varname[weight_variable]);
        WAIT; if (scale_check==SCALE_INTERRUPT) return;
    }
}

* * *

scales

Summary
int scales(data)
SURVO_DATA *data; /* pointer to data structure */

Description
The scales function removes all variables with the scale type '-' (no scale) from the list data->v of active variables.

data must be opened by data_open or data_open2.
scales is usually called after mask in statistical SURVO 84C modules to remove fields without scale from the analysis irrespective of the user’s selection. scales thus updates data->m_act and selection vector data->v.

Return Value
There is no return value.

See Also
mask
Example
int i;
SURVO_DATA dat;

i=data_open("TEST", &dat);
if (i<0) return;
mask(&dat); /* select variables according to MASK */
scales(&dat); /* remove variables without scale */
   * * *

shadow_create

Summary
int shadow_create(j)

int j;       /* edit line */

Description
The shadow_create function creates a shadow line consisting of \texttt{ed1} spaces for the \texttt{j}^{th} line (1\leq j\leq ed2) in the edit field. After the \texttt{shadow_create} call \texttt{zs}[j] is the index of the new shadow line.

If there is no more space for a new shadow line (\texttt{edshad} is the max. number), an error message \textit{Not space anymore for special display lines!} is displayed.

Return Value
shadow_create returns 1 if the shadow line has been created and -1 otherwise.

See Also
shadow_test
Example

```c
char x[LLENGTH];
int i;
for (i=0; i<c2; ++i) x[i]='7';
if (zs[10]==0)
    {
        i=shadow_create(10);
        if (i<0) return;
    }
edwrite(x,zs[10],1);
/* turns all characters on edit line 10 into reversed video (shadow value 7) */

***

shadow_test

Summary
int shadow_test(j)
int j; /* edit line */

Description
The shadow_test function frees the shadow line of the jth edit line if it consists of spaces (blanks) only.

Return Value
There is no return value.

See Also
shadow_create

Example
int j;
for (j=1; j<=r2; ++j)
    if (zs(j)>0) shadow_test(j);
/* frees all unnecessary shadow lines in the current edit field. */

***

SURVO.LIB
sp_init

Summary
int sp_init(editline)
int editline;   /* # of edit line */

Description
The sp_init function finds all the specifications from the subfield around editline and secondarily from the *GLOBAL* subfield.
sp_init forms the arrays char **spa,**spb consisting of the names (**spa on the left-hand side) and values (**spb on the right-hand side) of the specifications of the form
<name>=<value>
After the sp_init call the function spfind can be used to find the values of the specifications.

Return Value
sp_init returns 1 if there is enough space for all specifications. Otherwise -1 is returned.

See Also
spfind

Example
int i;

i=sp_init(r1+r-1);
   /* r1+r-1 is the current line in the edit field */
if (i<0) return;

***

spfind

Summary
int spfind(name)
char *name;   /* specification to be found */
Description

The `spfind` function searches for the specification `name` from the `**spa` list which has been created by the `sp_init` function earlier.

Return Value

If `name` is found, `spfind` returns the index (say `i`) of `name` in the `**spa` list. `spb[i]` is then the pointer to the value of `name`. If `name` is not found, -1 is returned.

See Also

`sp_init`

Example

```c
int i,k;
int x_home,y_home;
char x[LLENGTH]; *px[2];

i=sp_init(r1+r-1);    /* r1+r-1 is the current line in the edit field */
if (i<0) return;

i=spfind("HOME");
if (i>=0) {
    strcpy(x,spb[i]);
    k=split(x,px,2);
    if (k<0) {
        sprintf(sbuf,\"\nError in spec. HOME=%s\",spb[i]);
        sur_print(sbuf); WAIT; return;
    }
    x_home=atoi(px[0]);
    y_home=atoi(px[1]);
} else
    x_home=y_home=0;
```
split

Summary

int split(s,word,max)
char *s; /* null-terminated string */
char *word[]; /* pointers to words of string */
int max; /* max number of words to be found */

Description

The split function splits string s into tokens (words) word[0], word[1], ..., word[max-1] interpreting spaces and commas as delimiters. Since split writes an EOS character in place of every word ending with a space or a comma in s, the original contents of s are destroyed during the split call. After the call, the pointers word[0],word[1],... indicate the starting positions of the words in s.

Please note that the words will be destroyed if the contents of s are altered after the split call.

Return Value

split returns the number of words found which is max at most. Thus if the number of words in s is greater than max, the excessive words will not be found. There is no error return.

See Also

edread

Example

char x[]="PRINT 11,20";
char *word[3];
int i,k;

k=split(x,word,3);
for (i=0; i<k; ++i)
    printf("\nword[%d]=%s",i,word[i]);

prints:

word[0]=PRINT
word[1]=11
word[2]=20
Applications

*split* is the common tool when analyzing edit lines. A typical combination is, for example:

```c
edread(x,j);
k=split(x+1,word,10);
/* x+1=jth edit line without a control character */
```

---

sur_print

Summary

```c
int sur_print(string)
char *string;        /* null-terminated string */
```

Description

The *sur_print* function prints *string* in the window below the line defined by the global variable *scroll_line*. The output of *sur_print* will be scrolled automatically in that window.

*sur_print* is used mainly for temporary printouts with a 256 byte global string *sbuf*.

Return Value

There is no return value.

See Also

*write_string*

Example

```c
double result;
char str[LLENGTH];

fnconv(result,str,accuracy+2);
sprintf(sbuf,"\nResult=%s",str);
sur_print(sbuf);
```

---

SURVO.LIB
sur_wait

Summary
int sur_wait(time,display,break)
long time;        /* waiting time in ms */
int (*display)(); /* display function during wait */
int break;        /* 1: possible to break by any key */
/* 0: not possible to break */                  

Description
The sur_wait function creates a wait lasting time milliseconds. During
the wait the display function is called once every second (to indicate the
time elapsed, for example). If break=1, the wait can be interrupted by
pressing any key.

Return Value
sur_wait returns -1 if the wait has been interrupted by a key. Other-
wise 0 is returned.

Example
#include <stdio.h>
int sec=0;
main()
{
    extern seconds();
sur_wait(20000L,seconds,1);
}
seconds()
{
    printf(" %d",++sec);
}
/* This program counts to 20 seconds */
    * * *
tut_init, tut_end

Summary
int tut_init()
int tut_end()

Description
The tut_init function opens the tutorial file, if the current module is run under tutorial mode (system parameter etu>0).

  tut_init is called once immediately after s_init in those modules which operate at least partially in conversational mode (by using the prompt and nextch functions).

  Thus tut_init is not needed in modules which simply carry out their task without any prompts for the user. Error messages ending with WAIT do not require tut_init either.

  If tut_init has been called, the functions tut_end and s_end must be called (in this order) before the exit from the module.

Return Value
  There is no return value.

See Also
  s_init, s_end, prompt, nextch
Example
A typical construction in a SURVO 84C module is:

```c
#include "survo.h"
#include "survoext.h"

main(argc, argv)
int argc; char *argv[1];
{
    if (argc==1) return;
    s_init(argv[1]);
    tut_init();
    /*  ..........  */
    tut_end();
    s_end(argv[1]);
}
```

* * *

unsuitable

Summary
int unsuitable(data, j)
SURVO_DATA *data; /* pointer to data structure */
long *j; /* # of observation (record) */

Description
The unsuitable function tests whether observation j in data opened by data_open or data_open2 satisfies the restrictions imposed by IND and CASES specifications. Each module must call the conditions function once before the calls of unsuitable.

Return Value
unsuitable returns 1 if the conditions are not fulfilled and 0 otherwise.

See Also
conditions
Example

```c
int i;
long j;
SURVO_DATA dat;

i=data_open("TEST", &dat);
if (i<0) return;
i=conditions(&dat);
if (i<0) { data_close(&dat); return; }
for (j=dat.l1; j<=dat.l2; ++j)
{
    if (unsuitable(&dat, j)) continue;
    printf("%ld", j);
}
/* Numbers of observations satisfying
  the conditions are printed. */
```

***

varfind

Summary

```c
int varfind(data, name)
SURVO_DATA *data; /* pointer to data structure */
char *name; /* name of variable */
```

```c
int varfind2(data, name, error_display)
SURVO_DATA *data; /* pointer to data structure */
char *name; /* name of variable */
int error_display; /* 1=on 0=off */
```

Description

The varfind function finds the index corresponding to the given name of a variable (field) in data opened by data_open or data_open2.

Comparisons between names are performed by using the first 8 characters only. Trailing blanks are not counted. varfind is case-sensitive. Thus "Weight" is different from "weight".
Return Value

varfind returns # of variable or -1 if no variable corresponding to name is found. In the latter case an error message is displayed.

varfind2 works as varfind, but in case of an error (variable not found) no error message is displayed if error_display=0.

See Also

mask, activated

Example

int i;
SURVO_DATA dat;

i=data_open("TEST", &dat);
if (i<0) return;
i=varfind(&dat,"Weight");

wfind

Summary

int wfind(word1,word2,j)
char *word1; /* first word */
char *word2; /* second word */
int j; /* first edit line to be scanned */

Description

The wfind function searches the first line starting with the words word1 word2. The first line to be checked is j. Extra spaces before and between the words are not counted.

Return Value

wfind returns the index of the line and -1 if the line is not found.
Example
int k;
char name[]="ABC";
k=wfind("DATA",name,1);
finds the first line in the current edit field starting with the words DATA ABC.

***

write_string

Summary
int write_string(x,len,shadow,row,col)
char *x; /* string to be written */
int len; /* max # of bytes to be written */
char shadow; /* shadow (attribute) character */
int row,col; /* row and column of first character */

Description
The write_string function displays the len first bytes of x using the attribute given by shadow and starting from position (row,col).

Return Value
There is no return value.

See Also
sur_print

Example
A message for the user on the bottom line of the screen is produced typically by:
write_string(space,c3+8,' ',r3+2,1);
    /* Erase bottom line r3+2 */
write_string("Press any key!",14,'1',r3+2,1);
    /* Give message in 'red' */

***
8.2 Library SURVOMAT.LIB

Matrix functions are tools for making SURVO 84C operations for linear models, multivariate analysis, etc. For example, the matrix interpreter employs the library functions through the MAT operations.

The matrix operands and results are referred to by pointers of the double type. To enable dynamic space allocation, the matrices are always stored as one-dimensional arrays. The elements of the matrices (of type double) are saved columnwise. All computations are carried out in double precision.

For example, the function mat_transp is written as

```c
double *T, *X;
int m, n;
{
    register int i, j;
    for (i = 0; i < m; ++i) for (j = 0; j < n; ++j)
        T[j + n * i] = X[i + m * j];
    return(1);
}
```

It transposes a m*n matrix X and gives the result as a n*m matrix T. The elements of X are X[i + m * j] with row indices i = 0, 1, ..., m-1 and column indices j = 0, 1, ..., n-1.

The matrix functions do not allocate space for result matrices. For example, if the function above is called, space for the result T must have been reserved by

```c
T = (double *)malloc(m * sizeof(double));
if (T == NULL) { not_enough_space(); return(-1); }
```

If the matrix operation is successful, 1 is returned. Otherwise 0 or a negative integer is returned. In many cases the return value -i indicates that the operation has failed on row/column i of the matrix.
Matrix input and output

The SURVOMAT.LIB library does not support the matrix input and output directly. The SURVO.LIB library, however, includes the functions `matrix_load` and `matrix_save` functions for matrix files of the type used in MAT operations, for example. These functions should be used in all SURVO 84C operations which read and write matrices.

For example, the following SURVO 84C module transposes the matrix in matrix file A and saves the result in matrix file B, when the command

MTRANS A TO B

is given in SURVO 84C. The operation is equivalent to

\[ B = A' \]

The `matrix_load` call (28-29) also allocates space for matrix A and its row and column labels rlab,clab. Space is allocated for the transpose B on
lines 30-35 and after transposing the label of the matrix is updated on lines 37-38. Finally the matrix_save call (39-40) saves B in a matrix file. For additional information on matrix_load and matrix_save functions, see their descriptions in the SURVO.LIB library.

**Functions in library SURVOMAT.LIB**

survomat.h must be included for these functions.

**mat_add**

```c
int mat_add(T, X, Y, m, n)
double *T, *X, *Y;
int m, n;
```

computes \(T = X + Y\), where \(X\) and \(Y\) are \(m \times n\) matrices. mat_add always returns 1.

* * *

**mat_sub**

```c
int mat_sub(T, X, Y, m, n)
double *T, *X, *Y;
int m, n;
```

computes \(T = X - Y\), where \(X\) and \(Y\) are \(m \times n\) matrices. mat_sub always returns 1.

* * *

**mat_mlt**

```c
int mat_mlt(T, X, Y, m, n, r)
double *T, *X, *Y;
int m, n, r;
```

computes \(T = X \times Y\), where \(X\) is an \(m \times n\) and \(Y\) is an \(n \times r\) matrix. mat_mlt always returns 1.

* * *

---

**SURVOMAT.LIB**
mat_inv

int mat_inv(T, X, n, pdet)
double *T, *X;
int n;
double *pdet;

computes matrix $T$ as the inverse matrix of an $n \times n$ $X$ by the Gauss-Jordan elimination method. As a by-product, determinant of $X$ will be $pdet$. If any of the pivot elements are less than $1e-15$, $X$ is considered singular and no $T$ is computed; -$i$ will then be returned where $i$ is the current row index of the pivot element ($i=0, 1, \ldots, n-1$). In non-singular cases, 1 is returned. **Warning**: The matrix $X$ to be inverted is not preserved during the $\text{mat_inv}$ call.

* * *

mat_transp

int mat_transp(T, X, m, n)
double *T, *X;
int m, n;

transposes an $m \times n$ matrix $X$ to an $n \times m$ matrix $T$. $\text{mat_transp}$ always returns 1.

* * *

mat_mtm

int mat_mtm(T, X, m, n)
double *T, *X;
int m, n;

computes $T=X'X$, where $X$ is an $m \times n$ matrix. $\text{mat_mtm}$ always returns 1.

* * *

SURVOMAT.LIB
mat_mmt

int mat_mmt(T, X, m, n)
double *T, *X;
int m, n;

computes $T=XX'$, where $X$ is an $m \times n$ matrix. mat_mmt always returns 1.

* * *

mat_dmlt

int mat_dmlt(T, X, Y, m, n)
double *T, *X, *Y;
int m, n;

computes $T=X*Y$, where $X$ is an $m \times m$ diagonal matrix and $Y$ is an $m \times n$ matrix. mat_dmlt always returns 1.

* * *

mat_mltd

int mat_mltd(T, X, Y, m, n)
double *T, *X, *Y;
int m, n;

computes $T=X*Y$, where $X$ is an $m \times n$ matrix and $Y$ is a $n \times n$ diagonal matrix. mat_mltd always returns 1.

* * *
mat_center

int mat_center(T,X,m,n)
double *T,*X;
int m,n;

centers an \( m \times n \) matrix \( X \) by computing the means of the \( X \) columns as an \( n \) element \( T \) vector and subtracting them from the corresponding columns. \texttt{mat_center} always returns 1.

\*\*\*

mat_nrm

int mat_nrm(T,X,m,n)
double *T,*X;
int m,n;

normalizes the columns of an \( m \times n \) matrix \( X \) to length 1. The original column lengths (square root of sum of squares) will be stored as an \( n \) element vector \( T \). Columns of length=0 are not changed. \texttt{mat_nrm} always returns 1.

\*\*\*

mat_sum

int mat_sum(T,X,m,n)
double *T,*X;
int m,n;

computes the column sums of an \( m \times n \) matrix \( X \) as an \( n \) element vector \( T \). \texttt{mat_sum} always returns 1.

\*\*\*

SURVOMAT.LIB
**mat_chol**

```c
int mat_chol(T,X,n)
double *T,*X;
int n;
```

performs the Cholesky decomposition of an \(n \times n\) positive definite matrix \(X\). Hence an \(n \times n\) lower triangular matrix \(T\) satisfying \(X = TT'\) will be computed. If \(X\) is not positive definite, \texttt{mat_chol} returns \(-i\), where \(i\) (\(i = 0,1,\ldots,n-1\)) represents the column index where this assumption fails. If decomposition is successful, 1 is returned.

**mat_cholinv**

```c
int mat_cholinv(A,n)
double *A;
int n;
```

inverts an \(n \times n\) positive definite matrix \(A\) by the Cholesky method and writes the inverted matrix \(B\) partially on \(A\) according to the following scheme:

Before \texttt{mat_cholinv}: (Here \(n=5\) assumed)

\[
\begin{array}{cccccc}
0 & 1 & 2 & \ldots & n-1 & n \\
0 & a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & * \\
1 & a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & * \\
2 & a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & * \\
\ldots & a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & * \\
n-1 & a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & * \\
\end{array}
\]

After \texttt{mat_cholinv}:

\[
\begin{array}{cccccc}
0 & 1 & 2 & \ldots & n-1 & n \\
0 & a_{00} & b_{00} & b_{01} & b_{02} & b_{03} & b_{04} \\
1 & a_{10} & a_{11} & b_{11} & b_{12} & b_{13} & b_{14} \\
2 & a_{20} & a_{21} & a_{22} & b_{22} & b_{23} & b_{24} \\
\ldots & a_{30} & a_{31} & a_{32} & a_{33} & b_{33} & b_{34} \\
n-1 & a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & b_{44} \\
\end{array}
\]

Please note that the elements are assumed to be saved columnwise.
To have enough space for B, at least \(n^*(n+1)\) elements (of type double) must have been allocated for A before the `mat_cholinv` call.

If A is not positive definite, -i (where i is the first column dependent on previous ones) is returned. In a successful case 1 is returned.

** mat_cholmove

To overwrite A by its inverse completely, use `mat_cholmove(A,n)` after `mat_cholinv(A,n)` to obtain

\[
\begin{array}{cccccc}
0 & 1 & 2 & n-1 & n \\
0 & b_{00} & b_{02} & b_{03} & b_{04} \\
1 & b_{10} & b_{12} & b_{13} & b_{14} \\
2 & b_{20} & b_{22} & b_{23} & b_{24} \\
n-1 & b_{n-1,0} & b_{n-1,2} & b_{n-1,3} & b_{n-1,4} \\
\end{array}
\]

** mat_gram_schmidt

\[
\text{int mat_gram_schmidt(S,U,X,m,n,tol)}
\]

double *S,*U,*X;

int m,n;

double tol;

computes the Gram-Schmidt decomposition \(X=S*U\) for an \(m*n\) matrix X (with \(\text{rank}(X)=n\leq m\)), where \(S\) is an \(m*n\) matrix with orthonormal columns and \(U\) is \(n*n\) upper triangular.

The accuracy in checking the linear independency of columns is given by \(\text{tol}\). The value \(\text{tol}=1e-15\) is recommended.

Return value -i indicates that column i (\(i=0,1,\ldots,n-1\)) is linearly dependent on previous ones. After a successful decomposition, 1 is returned.

**
mat_p

int mat_p(X,n,k)
double *X;
int n,k;

transforms the \( n \times n \) matrix \( X \) by the pivotal operation by using the diagonal element \( k \) \((k=0,1,...,n-1)\) as the pivot.

* * *

mat_svd

int mat_svd(X,D,V,m,n,eps,tol)
double *X,*D,*V;
int m,n;
double eps,tol;

makes the singular value decomposition \( X=U \cdot \text{diag}(D) \cdot V' \) for an \( m \times n \) matrix \( X \) with \( m \geq n \). After the \texttt{mat_svd} call \( X \) will be overwritten by an \( m \times n \) matrix \( U \) which is columnwise orthogonal. \( D \) will be an \( n \) element vector consisting of singular values and \( V \) an \( n \times n \) orthogonal matrix.
\texttt{ eps} and \texttt{tol} are tolerance constants (See the source cited below). Suitable values are \texttt{eps=1e-16} and \texttt{tol=(1e-300)/eps}.

* * *

mat_tred2

int mat_tred2(d,e,A,n,tol)
double *d,*e,*A;
int n;
double tol;

reduces an \( n \times n \) symmetric matrix \( A \) to tridiagonal form using Householder's reduction. The diagonal of the result is stored as an \( n \) element vector \( d \) and the sub-diagonal as the last \( n-1 \) elements of an \( n \) element vector \( e \).
A will be overwritten by the transformation matrices. \( \text{tol} \) is an accuracy constant (see \text{mat_svd}).

Space for \( \text{d} \) and \( \text{e} \) (\( n \) elements each of type double) must be allocated before \text{mat_tred2} is called.

To get the eigenvalues and vectors after \text{mat_tred2}(\text{d}, \text{e}, \text{A}, n, \text{tol})\text{mat_tql2} has to be called.

\[ \star \star \star \]

\text{mat_tql2}

\begin{verbatim}
int mat_tql2(d,e,A,n,eps,maxiter)
double *d,*e,*A;
int n;
double eps;
int maxiter;

finds the eigenvalues and vectors of the \( n \times n \) tridiagonal matrix \( \text{A} \) obtained by \text{mat_tred2}. Matrix \( \text{A} \) will be overwritten by the eigenvectors and the eigenvalues will be saved in descending order as an \( n \) element vector \( \text{d} \).

\( \text{eps} \) in an accuracy constant (see \text{mat_svd}). Maximum number of iterations for one eigenvalue is \( \text{maxiter} \). \( \text{maxiter}=30 \) is recommended. In case of no convergence within \( \text{maxiter} \) iterations, \(-1\) is returned. If the eigenvalues and vectors are obtained, \( 1 \) is the return value.

\text{mat_tred2} and \text{mat_tql2} have been written using the ALGOL procedures \text{tred2} and \text{tql2} as the basis. See \text{Handbook for Automatic Computation}, Volume II, edited by J.H.Wilkinson and C.Reinsch, (Springer 1971).
\end{verbatim}

\[ \star \star \star \]

\text{solve_upper, solve_lower, solve_diag}

\begin{verbatim}
int solve_upper(X,A,B,m,k,eps)
double *X,*A,*B;
int m,k;
double eps;

solves the system of linear equations \( \text{A}X=B \) where \( \text{A} \) is an \( m \times m \) upper triangular matrix and \( \text{B} \) is an \( m \times k \) matrix. Before calling \text{solve_upper}, space must also be allocated to the \( m \times k \) solution matrix \( \text{X} \).

If any of the pivot elements is smaller than \( \text{eps} \), \text{solve_upper} returns \(-1\).
\end{verbatim}
where \( i=0,1,...,m-1 \) is the current column. After a successful solution, 1 is returned.

- `solve_lower` works as `solve_upper` but with an \( m\times m \) lower triangular matrix \( A \).
- `solve_diag` works as `solve_upper` but with an \( m\times m \) diagonal matrix \( A \).

---

### solve_symm

```c
int solve_symm(X, A, B, m, k, eps) double *X, *A, *B; int m, k; double eps;
```
solves the system of linear equations \( A X = B \) where \( A \) is an \( m \times m \) positive definite matrix and \( B \) is an \( m \times k \) matrix. Before calling `solve_symm`, space must also be allocated to the \( m \times k \) solution matrix \( X \).

If any of the pivot elements is smaller than \( \text{eps} \), `solve_symm` returns \(-i\), where \( i=0,1,...,m-1 \) is the current column. After a successful solution, 1 is returned. If \( A \) is not positive definite, `solve_symm` calls `ortholin1`.


---

### ortholin1

```c
int ortholin1(A, n, m, B, k, eps, X, improvement) double *A; int n, m; double *B; int k; double eps; double *X; int improvement;
```
gives least squares solutions for \( A X = B \), where \( A \) is an \( n \times m \) matrix, \( B \) an \( n \times k \) matrix and \( n \geq m \).

`eps` is the maximal relative rounding error (typically \( \text{eps}=1e-15 \)).

---

**sis_tulo**

```c
double sis_tulo(a,b,sa,sb,n)
double *a,*b;
int sa,sb,n;

is an assembler routine (written by Timo Patovaara) for computation of
the inner product

\[
+ a[(n-1)*sa] * b[(n-1)*sb]
\]

To speed up computations, many of the SURVOMAT.LIB functions use
**sis_tulo** for scalar products.

**survomat.lib**
8.3 Library DISTRIB.LIB

Many statistical operations give test statistics with appropriate $P$ values obtained from standard distributions. To provide such $P$ values and other numerical characteristics related to theoretical distributions, a set of C routines for density, distribution and inverse distribution functions of the common continuous distributions have been written by T. Patovaara. These functions are presented in the DISTRIB.LIB library.

The sources for the algorithms used are:

Abramowitz and Stegun: *Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables*, Dover 1970.

**Functions in library DISTRIB.LIB**

**cdf_std**

```c
double cdf_std(x)
double x;
returns the cumulative distribution function of the standardized normal distribution with the accuracy of the machine.
```

***

**inv_std**

```c
double inv_std(p)
double p;
returns $x = \text{inv}_F(p)$ for a given value of $p$ ($0 < p < 1-1.0E-15$), where $\text{inv}_F$ is the inverse distribution function of the standardized normal distribution.
```
Accuracy:
- $0 < p \leq 1\times10^{-4}$: the accuracy of the machine
- $1\times10^{-4} < p \leq 1\times10^{-8}$: 13-10 significant digits
- $1\times10^{-8} < p \leq 1\times10^{-11}$: 9-5 significant digits
- $1\times10^{-11} < p \leq 1\times10^{-15}$: 4-2 significant digits

**pdf_t**

```c
double pdf_t(x, n)
double x, n;
```

returns the Student’s density function for a value $x$ with $n$ ($n > 0$) degrees of freedom with the accuracy of the machine.

**cdf_t**

```c
double cdf_t(x, n)
double x, n;
```

returns the cumulative distribution function of the Student’s distribution for a value $x$ with $n$ ($n > 0$) degrees of freedom.

Accuracy: 10-14 significant digits for $|x| \geq 1\times10^{-7}$.
**inv_t**

```c
double inv_t(p, n) double p, n;
```

returns \( x = \text{inv}_F(p, n) \) for a given value of \( p \) (\( 0 < p \leq 1\times10^{-15} \)), where \( \text{inv}_F \) is the inverse distribution function of the Student’s distribution for \( n \) (\( n > 0 \)) degrees of freedom.

Accuracy:
- \( 0.5 + 1\times10^{-4} \leq p < 1\times10^{-7} \): over 10 significant digits
- \( 1\times10^{-7} \leq p < 1\times10^{-9} \): 10-9 significant digits
- \( 1\times10^{-9} \leq p < 1\times10^{-12} \): 8-5 significant digits
- \( 1\times10^{-12} \leq p \leq 1\times10^{-15} \): 4-2 significant digits

Similar accuracy for \( 0 < p < 0.5 \).

* * *

**pdf_ch2**

```c
double pdf_ch2(x, n) double x, n;
```

returns the \( \chi^2 \) density function for a value \( x \) with \( n \) (\( n > 0 \)) degrees of freedom with the accuracy of the machine.

* * *

**cdf_ch2**

```c
double cdf_ch2(x, n, rel_error) double x, n, rel_error;
```

returns the cumulative distribution function of the \( \chi^2 \) distribution for a value \( x \) with \( n \) (\( n > 0 \)) degrees of freedom.

Accuracy: determined by \( \text{rel}_\text{error} \) (\( 1\times10^{-15} \leq \text{rel}_\text{error} < 0.5 \)).

* * *

---

**DISTRIB.LIB**
inv\_chi2

double inv\_chi2(p,n)
double p,n;

returns \( x = \text{inv}\_F(p,n) \) for a given value of \( p \) (1E-6 ≤ \( p < 1\-1E-6 \)), where \( \text{inv}\_F \) is the inverse distribution function of the \( \chi^2 \) distribution for \( n \) (\( n > 0 \)) degrees of freedom.
Accuracy: over 10 significant digits.

* * *

pdf\_beta

double pdf\_beta(x,a,b)
double x,a,b;

returns the Beta density function for a value \( x \) and parameters \( a,b \) (\( a,b > 0 \)) with the accuracy of the machine.

* * *

cdf\_beta

double cdf\_beta(x,a,b,rel\_error)
double x,a,b,rel\_error;

returns the cumulative distribution function of the Beta distribution for a value \( x \) and parameters \( a,b \) (\( a,b > 0 \)).
Accuracy: determined by \( \text{rel}\_error \) (1E-15 ≤ \( \text{rel}\_error \) < 0.5).

* * *
**inv_beta**

double inv_beta(p, a, b, s_digits)
double p, a, b;
int s_digits;

returns \( x = \text{inv}_F(p, a, b) \) for a given value of \( p \) (\( 0 < p \leq 1-1E-15 \)), where inv_F is the inverse distribution function of the Beta distribution with parameters \( a, b \).

Accuracy: The number of significant digits is determined by \( s\_digits \) (\( 2 \leq s\_digits \leq 14 \)).

* * *

**pdf_f**

double pdf_f(x, n1, n2)
double x, n1, n2;

returns the F density function for a value \( x \) and \( n_1 \) and \( n_2 \) (\( n_1, n_2 > 0 \)) degrees of freedom with the accuracy of the machine.

* * *

**cdf_f**

double cdf_f(x, n1, n2, rel_error)
double x, n1, n2, rel_error;

returns the cumulative distribution function of the F distribution for a value \( x \) and \( n_1 \) and \( n_2 \) (\( n_1, n_2 > 0 \)) degrees of freedom.

Accuracy: same as in t distribution if \( n_1 = 1 \) or \( n_2 = 1 \). Otherwise determined by \( rel\_error \) (\( 1E-15 \leq rel\_error < 0.5 \)).

* * *
inv_f

double inv_f(p,n1,n2,s_digits)
double p,n1,n2;
int s_digits

returns $x = \text{inv}_F(p,n_1,n_2)$ for a given value of $p$ ($0 < p < 1$), where \text{inv}_F is the inverse distribution function of the F distribution for $n_1$ and $n_2$ ($n_1,n_2 > 0$) degrees of freedom.

Accuracy: same as in t distribution if $n_1 = 1$ or $n_2 = 1$.

Otherwise the number of significant digits is determined by $s\_digits$ ($2 \leq s\_digits \leq 14$).

***

lg_gamma

double lg_gamma(x)
double x;

returns the natural logarithm of the gamma function with the accuracy of the machine.

***

DISTRIB.LIB
## Index to SURVO 84C library functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
<th>Page</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>activated</td>
<td>9,22</td>
<td>mat_mmt</td>
<td>71</td>
</tr>
<tr>
<td>cdf_beta</td>
<td>82</td>
<td>mat_mtm</td>
<td>70</td>
</tr>
<tr>
<td>cdf_chi2</td>
<td>81</td>
<td>mat_nrm</td>
<td>72</td>
</tr>
<tr>
<td>cdf_f</td>
<td>83</td>
<td>mat_p</td>
<td>75</td>
</tr>
<tr>
<td>cdf_std</td>
<td>79</td>
<td>mat_sub</td>
<td>69</td>
</tr>
<tr>
<td>cdf_t</td>
<td>80</td>
<td>mat_sum</td>
<td>72</td>
</tr>
<tr>
<td>conditions</td>
<td>9,13,22</td>
<td>mat_svd</td>
<td>75</td>
</tr>
<tr>
<td>create_newvar</td>
<td>23</td>
<td>mat_tql2</td>
<td>76</td>
</tr>
<tr>
<td>data_alpha_load</td>
<td>24</td>
<td>mat_transp</td>
<td>67,70</td>
</tr>
<tr>
<td>data_close</td>
<td>25</td>
<td>mat_tred2</td>
<td>75</td>
</tr>
<tr>
<td>data_load</td>
<td>13,26</td>
<td>nextch</td>
<td>47</td>
</tr>
<tr>
<td>data_open</td>
<td>8,19,27</td>
<td>ortholin1</td>
<td>77</td>
</tr>
<tr>
<td>data_open2</td>
<td>29</td>
<td>output_close</td>
<td>48</td>
</tr>
<tr>
<td>data_save</td>
<td>30</td>
<td>output_line</td>
<td>14,48</td>
</tr>
<tr>
<td>edline2</td>
<td>8,18,31</td>
<td>output_open</td>
<td>13,48</td>
</tr>
<tr>
<td>edread</td>
<td>15,32</td>
<td>pdf_beta</td>
<td>82</td>
</tr>
<tr>
<td>edwrite</td>
<td>15,33</td>
<td>pdf_chi2</td>
<td>81</td>
</tr>
<tr>
<td>empty_line</td>
<td>34</td>
<td>pdf_f</td>
<td>83</td>
</tr>
<tr>
<td>fconv</td>
<td>35</td>
<td>pdf_t</td>
<td>80</td>
</tr>
<tr>
<td>fi_create</td>
<td>36</td>
<td>prompt</td>
<td>49</td>
</tr>
<tr>
<td>fnconv</td>
<td>14,37</td>
<td>rem_pr</td>
<td>8,39</td>
</tr>
<tr>
<td>hae_apu</td>
<td>38</td>
<td>scales</td>
<td>11,54</td>
</tr>
<tr>
<td>initRemarks</td>
<td>8,39</td>
<td>scale_check</td>
<td>11</td>
</tr>
<tr>
<td>inv_beta</td>
<td>83</td>
<td>scale_ok</td>
<td>11,52</td>
</tr>
<tr>
<td>inv_chi2</td>
<td>82</td>
<td>shadow_create</td>
<td>18,55</td>
</tr>
<tr>
<td>inv_f</td>
<td>84</td>
<td>shadow_test</td>
<td>18,56</td>
</tr>
<tr>
<td>inv_std</td>
<td>79</td>
<td>sis_tulo</td>
<td>78</td>
</tr>
<tr>
<td>inv_t</td>
<td>81</td>
<td>solve_diag</td>
<td>76</td>
</tr>
<tr>
<td>lastline2</td>
<td>40</td>
<td>solve_lower</td>
<td>76</td>
</tr>
<tr>
<td>lg_gamma</td>
<td>84</td>
<td>solve_symm</td>
<td>77</td>
</tr>
<tr>
<td>mask</td>
<td>9,40</td>
<td>solve_upper</td>
<td>76</td>
</tr>
<tr>
<td>matrix_format</td>
<td>41</td>
<td>sp_init</td>
<td>9,57</td>
</tr>
<tr>
<td>matrix_load</td>
<td>42,68</td>
<td>spfind</td>
<td>9,57</td>
</tr>
<tr>
<td>matrix_print</td>
<td>44</td>
<td>split</td>
<td>17,59</td>
</tr>
<tr>
<td>matrix_save</td>
<td>45,68,69</td>
<td>sur_print</td>
<td>11,60</td>
</tr>
<tr>
<td>mat_add</td>
<td>69</td>
<td>sur_wait</td>
<td>61</td>
</tr>
<tr>
<td>mat_center</td>
<td>72</td>
<td>s_end</td>
<td>16,17,50</td>
</tr>
<tr>
<td>mat_chol</td>
<td>73</td>
<td>s_init</td>
<td>7,15,51</td>
</tr>
<tr>
<td>mat_cholinv</td>
<td>73</td>
<td>tut_end</td>
<td>62</td>
</tr>
<tr>
<td>mat_cholmove</td>
<td>74</td>
<td>tut_init</td>
<td>62</td>
</tr>
<tr>
<td>mat_dmlt</td>
<td>71</td>
<td>unsuitable</td>
<td>9,12,63</td>
</tr>
<tr>
<td>mat_gram_schmidt</td>
<td>74</td>
<td>varfind</td>
<td>64</td>
</tr>
<tr>
<td>mat_inv</td>
<td>70</td>
<td>waitRemarks</td>
<td>8,39</td>
</tr>
<tr>
<td>mat_mlt</td>
<td>69</td>
<td>wfind</td>
<td>65</td>
</tr>
<tr>
<td>mat_mltd</td>
<td>71</td>
<td>write_string</td>
<td>66</td>
</tr>
</tbody>
</table>