3D/2D modelling suite for integral water solutions

DELFT3D

Deltares systems

NEFIS Viewer Selector

Deltares

Enabling Delta Life

Programmers Manual

DRAFT

ViewerSelector

Interaction with other processes and how to build such processes

Programmer's Manual

Version: 1.23
Revision: 78359

5 November 2026

ViewerSelector, Programmer's Manual

DRAFT

Published and printed by:
Deltares
Boussinesqweg 1
2629 HV Delft
P.O. 177
2600 MH Delft
The Netherlands

telephone: +31 88 335 82 73
e-mail: Information
www: Deltares

For sales contact:
telephone: +31 88 335 81 88
e-mail: Sales
www: Sales & Support

For support contact:
telephone: +31 88 335 81 00
e-mail: Support
www: Sales & Support

Copyright © 2026 Deltares
All rights reserved. No part of this document may be reproduced in any form by print, photo print, photo copy, microfilm or any other means, without written permission from the publisher: Deltares.

Contents

1 Introduction **2**

2 Communication with external processes **3**
2.1 ViewerSelector variable . . . 3

3 Building your own external processes **4**
3.1 EXEC process . . . 4
3.2 EXEC process PAR parameter 4
3.3 EXEC process WITH variable 5
3.4 EXEC process RETN . . . 6

4 Examples **8**
4.1 Echo parameters . . . 8
4.2 Write arguments to `stderr` . . . 8
4.3 Read variable from `stdin` . . . 8
4.4 Data passed to Fortran function . . . 9
4.5 Create variable and write it to `stdout` . . . 10

References **11**

A Appendix **12**
A.1 Convenience Functions . . . 12
A.2 ”vscf.h” . . . 13
A.3 C-Fortran interfacing . . . 14
A.3.1 Argument passing . . . 14
A.3.2 External names . . . 15
A.3.3 Return values . . . 15

DRAFT

Preface

The ViewerSelector is a tool for inspecting NEFIS files and selecting data from such files. One of the options of this tool is to pass data to and from external processes. So there is a need to know how to build such processes.

This manual describes how the ViewerSelector interacts with such processes and how to build such processes.

DRAFT

1 Introduction

This manual describes how to create processes to be used from the EXEC-command of the ViewerSelector. Processes can use data provided by the ViewerSelector and/or return data to the ViewerSelector. So these kind of processes can be used for instance to filter data, display data in a graphical form (TEKAL or another package) and so on.

Chapter 2 describes how it works and Chapter 3 gives details by exploring a lot of examples.

This manual is meant to be used by programmers who have a common knowledge of the ViewerSelector and NEFIS. Some experience in C-programming is preferred, because the main program of most processes must be written in C.

DRAFT

2 Communication with external processes

The ViewerSelector can communicate with external processes. An external process can read VS-variables from stdin and may write VS-variables to stdout. The ViewerSelector can also pass parameters to an external process.

The ViewerSelector activates external processes through the EXEC-command.

2.1 ViewerSelector variable

In this manual the term ViewerSelector variable will be used very often. This variable is described in a structure `St_var` and can be found in file <vscf.h> (see Appendix A.2). The element of this structure have the same names as described in the NEFIS User Manual, NEFIS UM (2013), and will not be explained in this document. One exception is the element `varpnt`. This is a pointer to the object which contains the values of the variable.

DRAFT

3 Building your own external processes

One of the features of the ViewerSelector is that you can build your own processes, which can be activated from the ViewerSelector. Such a process can be a simple executable or a script file which combines, by means of pipes, more than one executable.

To make live easier, a library with convenience functions and a c-header file are also provided. A description of the functions in this library is found in Appendix A.1.

The rest of this Chapter contains examples of the EXEC-command and an explanation of the processes, used in these commands.

3.1 EXEC process

This is the most simple form of the `exec` command. It simply executes the specified process. This process needs no data from the ViewerSelector and will not return data to ViewerSelector. It also needs no parameters. You can not start processes which use `stdin`!!! As an example you can type:

```
EXEC date
```

If for instance, you want to execute `vi` from the ViewerSelector (which uses stdin) type:

```
!vi
```

3.2 EXEC process PAR parameter ...

The specified process can either be a script (or <*.bat> file) or an executable program.

Example 4.1 is a simple script which echo's the specified parameters to `stdout`. This script must be executable and there must be a path to this script. You can call this script with the command:

```
EXEC exa01 PAR '10' john '45.56' charlie
```

The output would be:

The parameters are: `10 john 45.56 charlie`

In this script file you can of course execute one or more executables or other scripts, which use these parameters.

If you want to execute such a process in background (Linux only), you can specify a parameter (the last) '&'.

Example 4.2 is the C-source for an executable, which takes all it's run time parameters and prints them to `stdout`. You can execute this program `exa02`, after compiling and linking of course, with the ViewerSelector command:

```
EXEC exa02 par John and Mary
```

The output would be:

```
The parameters of program exa02 are:  John and Mary
```

DRAFT

3.3 EXEC process WITH variable ...

If you start a process this way, the ViewerSelector writes one or more variables to `stdin` of the specified process. The variables are written in a special way and can only be retrieved one by one (first in, first out) by the specified process using the convenience function `VS_read_var_from_pipe` (see explanation of this function in Appendix A.1).

Example 4.3 contains a C-source for a program that reads this variable from `stdin`.

Explanation
line 8 Define `p` as a pointer to a structure `St_var`. This structure is defined in <vscf.h>

line 14 using the function `VS_read_var_from_pipe`, the first variable, supplied by the ViewerSelector will be read from `stdin`. A second call to this function would try to read the second variable, and so on.

line 16-18 If `p` still contains the value `NULL`, something went wrong during reading of the variable, so send a message to `stderr` and exit program.

line 22-25 Check the type of variable, using the function `VS_get_data_type`. This program can only handle type `FLOAT` (for types, see also <vscf.h>). If the type is wrong, send a message to `stderr` and exit program.

line 30-33 If everything is okey, write the values of this variable to `stdout`. The number of values can be retrieved using the function `VS_calc_number_of_values`. Line 33 contains a little bit complicated type casting, caused by the fact that `p->varpnt` is of type void.

line 36 Exit the program with exit code 0.

After building this program you can, for example, test it as follows:

```
LET a = 12.25 EXEC exa03 PAR a
```

The output will be:

```
12.25
```

This example is extended in example 4.4. In this example the data will be passed to a Fortran function, and printed there.

Explanation
file: <exa04.c>

line 5 This line contains the prototype of the Fortran function.

line 7-8 This line contains the definition of a macro FORFUN. This macro is used to call the Fortran function forfun on line 36. This function expects two parameters (see file: <forfun.f>), an array with floats (`REAL*4`), and an `INTEGER*4`, containing the length of the array. Because Fortran expects all parameters to be references, the address of the second parameter is passed (`&p2`). The first parameter is assumed to be a reference. Defining a macro for Fortran functions is a good idea, because interfacing from C to Fortran can be different for other machines.

DRAFT

Remark:
- ◇ It's a good idea to put the prototypes and macro's in a header file.

line 9 to 35 See previous example

line 36 Here the Fortran function is called, by using the macro FORFUN.

file:<forfun.f>

line 1-10 This subroutine prints the contents of the array to `stdout`.

After building this program, it will give the same result as the previous example.

3.4 EXEC process RETN

If a process is executed from the ViewerSelector this way, the process will be started and the ViewerSelector will try to read new variables, written to `stdout` by the specified process. Writing variables to `stdout` is done using the function `VS_write_var_to_pipe` (see Appendix A.1). These type of processes should not use `stdout` for other purposes!

Example 4.5 is an example of a program that creates a variable, assigns some values to this variable and then writes this variable to `stdout`.

Explanation
line 14-16 Create space for a new variable, using the function `VS_create_new_var`. The new variable will have the name `New`, type `float` (i.e `REAL*4`). The number of group dimensions will be 1 and the number of element dimensions also 1. The length of the first group dimension is 10. So we create space for a variable of 10 floats.

line 18-20 Error handling.

line 24-25 Here we specify the group name and element name.

line 28-29 Fill variable with values 1-10.

line 33 Write variable to `stdout`. This variable will be read by the ViewerSelector.

Executing the following commands from the ViewerSelector will give as a result:

```
EXEC exa05 RETN
DISP MEMO
Variable New from ....derived(10) ELEMENT from exa05(1) REAL *4
WRITE New
** VARIABLE New from ....derived(10) ELEMENT from exa05(1) REAL *4
**    NROW       NCOL     NPLANE
        10          1          1

**   PLANE: 1

**  COLUMN:              1
**     ROW
         1   1.0000000e+00
         2   2.0000000e+00
         3   3.0000000e+00
         4   4.0000000e+00
         5   5.0000000e+00
         6   6.0000000e+00
```

```
 7   7.0000000e+00
 8   8.0000000e+00
 9   9.0000000e+00
10   1.0000000e+01
```

DRAFT

4 Examples

This chapter contains the texts of the examples, used in previous Chapters. All examples are printed with line numbers and are no part of the examples.

4.1 Echo parameters

This example is a simple shell script, named `exa01`, which simple echo's it's parameters to `stdout`.

```
1   #! /bin/ksh
2   #
3   # echo all parameters to stdout
4   #
5   echo the parameters are: \$*
6
7   exit 0
```

4.2 Write arguments to `stderr`

This is a list of file <exa02.c>. This program simply takes all it's arguments and writes them to `stdout`.

```
 1   #include <stdio.h>
 2   main( int argc, char * argv[] )
 3   {
 4       int i;
 5
 6       (void)printf("The parameters in program %s are: ", argv[0] );
 7
 8       for (i = 1 ; i < argc ; i++ ) {
 9           (void)printf ("%s ", argv[i] );
10       }
11       printf ("\n");
12
13       return 0;
14
15   }
```

4.3 Read variable from `stdin`

This is a list of file <exa03.c>

```
 1   #include <stdio.h>
 2   #include <strings.h>
 3   #include "vscf.h"       /* functions and types for VS functions */
 4
 5
 6   main(int argc, char * argv[])
 7   {
 8       struct St_var * p;          /* vs variable */
 9       int    i;
10       int    nvars;
11
12
13       /* Read variable from stdin */
14       p = VS_read_var_from_pipe ( );
15
16       if (p == NULL) {
17           (void)fprintf ( stderr, " Error reading pipe\n" );
18           return 1;
19       }
20
21       /* Check the data type, must be FLOAT (i.e. REAL*4 */
```

DRAFT

```
22      if ( VS_get_data_type( p ) != FLOAT ) {
23          (void)fprintf ( stderr, " Variable %s has wrong type\n" ,
24                                          p >varnam );
25          return 1;
26      }
27
28
29      /* Print the values */
30      nvars = VS_calc_number_of_values ( p );
31
32      for ( i = 0 ; i < nvars ; i++ ) {
33          (void)printf( "%f\n", ((float *)p>varpnt)[i] );
34      }
35
36      return 0;
37  }
```

4.4 Data passed to Fortran function

This is a list of file <exa04.c> and <forfun.f>

```
 1  #include <stdio.h>
 2  #include <strings.h>
 3  #include "vscf.h"       /* functions and types for VS functions */
 4
 5  void forfun ( float *, long * );
 6
 7  #define FORFUN(p1,p2) \
 8      (void)forfun(p1,&p2)
 9
10
11  main(int argc, char * argv[])
12  {
13      struct St_var * p;          /* vs variable */
14      long    nvars;
15
16
17      /* Read variable from stdin */
18      p = VS_read_var_from_pipe ( );
19
20      if (p == NULL) {
21          (void)fprintf ( stderr, " Error reading pipe\n" );
22          return 1;
23      }
24
25      /* Check the data type, must be FLOAT (i.e. REAL*4 */
26      if ( VS_get_data_type( p ) != FLOAT ) {
27          (void)fprintf ( stderr, " Variable %s has wrong type\n" ,
28                                          p>varnam );
29          return 1;
30      }
31
32
33      /* Print the values */
34      nvars = VS_calc_number_of_variables ( p );
35
36      FORFUN ( p>varpnt, nvars );
37
38      return 0;
39  }
```

This is a list of file <forfun.f>.

```
1         SUBROUTINE FORFUN ( ARRAY, LARRAY )
```

DRAFT

```
 2
 3         REAL    ARRAY(*)
 4         INTEGER LARRAY
 5
 6         DO 10 I = 1, LARRAY
 7            WRITE (*,*) ARRAY(I)
 8      10 CONTINUE
 9
10         END
```

4.5 Create variable and write it to `stdout`

This is a list of file <exa05.c>.

```
 1   #include <stdio.h>
 2   #include <string.h>
 3   #include "vscf.h"       /* functions and types for VS functions */
 4
 5   main(int argc, char * argv[])
 6   {
 7       struct St_var * p;          /* vs variable */
 8       long            grpdms[5];
 9       long            elmdms[5];
10       int             i;
11
12
13       /* Create space for new Variable */
14       elmdms[0] = 1;
15       grpdms[0] = 10;
16       p = VS_create_new_var ( "New", "REAL", 4, 1, grpdms, 1, elmdms );
17
18       if (p == NULL) {
19           (void)fprintf ( stderr, " Error creating variable\n");
20           return 1;
21       }
22
23       /* Use grpnam and elmnam to specify the origin */
24       (void)strcpy ( p>grpnam, "....derived" );
25       (void)strcpy ( p>elmnam, "from exa05" );
26
27       /* Fill variable with values 1 .. 10 */
28       for (i = 0; i < grpdms[0] ; i++ ) {
29           ((float *)p>varpnt)[i] = (float)i+1;
30       }
31
32       /* Write variable to stdout */
33       VS_write_var_to_pipe ( p );
34
35       return 0;
36   }
```

DRAFT

References

NEFIS UM, 2013. *NEFIS User Manual*. Deltares, 5.00 ed.

DRAFT

A Appendix

A.1 Convenience Functions

Name: `VS_malloc` - memory allocator

Synopsis:
```
#include "vscf.h"
void * VS_malloc ( size_t size );
```

Description: Returns a pointer to space for an object of size size, or NULL if the request cannot be satisfied.

Name: VS_free - Memory deallocator

Synopsis:
```
#include "vscf.h"
void VS_free ( void * p );
```

Description: Deallocates the space pointed to by `p`; it dose nothing if `p` is `NULL`. `p` must be a pointer to space previously allocated by `VS_malloc`.

Name: VS_create_new_var - Create space for ViewerSelector variable

Synopsis:
```
#include "vscf.h"
struct St_var VS_create_new_var ( char * varnam, char * elmtyp,
long nbytsg, long grpndm, long grpdms[], long elmndm, long
elmdms[] );
```

Description: This function creates space for a structure `St_var` and space for the values. The elements `varnam`, `elmtyp`, `nbytsg`, `grpndm`, `grpdms`, `elmndm`, `elmdms`, `varpnt` of this structure are assigned a value. This function returns a pointer to this structure or `NULL`, in case of errors (for instance: not enough memory). All parameters are input parameters. See NEFIS User Manual NEFIS UM (2013) for their description.

Name: `VS_calc_number_of_values` - calculate number of values of a variable

Synopsis:
```
#include "vscf.h"
size_t VS_calc_number_of_values ( struct St_var * p );
```

Description: Using the elements `elmdms` and `grpdms` from structure `p`, this function calculates the number of values of this variable.

Name: `VS_read_var_from_pipe` - Read a ViewerSelector variable from pipe

Synopsis:
```
#include "vscf.h"
struct St_var * VS_read_var_from_pipe ( void );
```

Description: This function tries to read a ViewerSelector variable from `stdin`. This variable must be written to this file from another process (or the ViewerSelector) with the function `VS_write_var_to_pipe`. A second call to this function will try to read another variable, and so on. It returns a pointer to a structure `St_var` or `NULL` in case of error.

Name: `VS_write_var_to_pipe` - write ViewerSelector variable to pipe

Synopsis:
```
#include "vscf.h"
void VS_write_var_to_pipe ( struct St_var * p );
```

Description: This function writes a ViewerSelector variable, pointed to by `p`, to `stdout`. These variables can be retrieved in another process (or the ViewerSelector) using the function `VS_read_var_from_pipe`. Processes, using this function should not use `stdout` for other purposes.

Name: `VS_get_data_type` - returns data type of ViewerSelector variable

Synopsis:
```
#include "vscf.h"
int VS_get_data_type ( struct St_var * p ) ;
```

Description: This function returns the data type of variable pointed to by `p`. The return value can be compared with the storage types as defined in file <vscf.h>.

DRAFT

A.2 ”vscf.h”

Print out of the file <vscf.h> footnotesize

```
/*          Copyright (C) 1993 Delft Hydraulics

 System      : ViewerSelector

 $Header: /u/cvsroot/viewsel/doc/vs pm.001,v 1.4 1993/11/22 14:16:30 hoeks_a E

 Programmer  : Abe Hoekstra   CSO
 Part        : ViewSel

 $Log: vs pm.001,v $
 * Revision 1.2  1993/07/06  09:34:33  hoeks_a
 * Development step
 *
 * Revision 1.1  1993/07/05  07:21:35  hoeks_a
 * Initial check in
 *

 @begin@
 @end@
 */

#ifndef _VSCF_INCLUDED
#   ifndef MSDOS
#       ifndef LINT
            static char VSCF_rcsid[] = "$Id: vs_pm_appendix.tex 50387 2017-03-
#       endif
#   endif


    struct St_var{
        char           varnam[17];
        char           elmtyp[9];
        long           nbytsg;
        char           elmqty[17];
        char           elmunt[17];
        char           elmdes[65];
        long           nintat;
        char           intatn[5][17];
        long           intatv[5];
        long           nreaat;
        char           reaatn[5][17];
        float          reaatv[5];
        long           nstrat;
        char           stratn[5][17];
        char           stratv[5][17];
        char           grpnam[17];
        long           grpndm;
        long           grpdms[5];
        char           elmnam[17];
        long           elmndm;
        long           elmdms[5];
        void          *varpnt;
        struct St_var *next;
    };

    enum STORAGE_TYPE {DBLCMPL, COMPLEX, DOUBLE, FLOAT, INT, SHORT,
                       BOOL, BOOLSHRT, CHAR, UNDEF} ;

#   if defined(__STDC__) || defined(MSDOS)
    extern struct St_var * VS_create_new_var ( char *, char *,
                                long, long, long[], long, long[] );
    extern size_t          VS_calc_number_of_values (
```

DRAFT

```
63                                      struct St_var * );
64      extern struct St_var * VS_read_var_from_pipe( void );
65      extern int             VS_write_var_to_pipe( struct St_var * );
66      extern int             VS_get_data_type ( struct St_var * );
67      extern void          * VS_malloc ( size_t );
68      extern void            VS_free ( void * );
69  #   else /* not __STDC__ */
70      extern struct St_var * VS_create_new_var ();
71      extern size_t          VS_calc_number_of_values ()
72      extern struct St_var * VS_read_var_from_pipe();
73      extern int             VS_write_var_to_pipe();
74      extern int             VS_get_data_type ();
75      extern void          * VS_malloc ();
76      extern void            VS_free ();
77  #   endif /* __STDC__ || MSDOS */
78
79
80  #   define _VSCF_INCLUDED
81  #endif
```

A.3 C-Fortran interfacing

C-Fortran interfacing differs from machine to machine or from compiler to compiler, although the basic principals are almost the same. The following points need attention:

- ◇ Argument passing
- ◇ External names
- ◇ Return value

A.3.1 Argument passing

Fortran expects all arguments to be call by reference. So what you must do if you want to pass arguments from C to Fortran, be sure that all data is passed by reference.

Example:Fortran subroutine: footnotesize

```
SUBROUTINE SORT ( IARRAY, LARRAY )
INTEGER IARRAY(LARRAY), LARRAY
```

Call from C: footnotesize

```
void sort ( long *, long * ) /* prototype of fortran function */
..
..
main() {
..
..
    long iar[100];
    long lar;
..
    void sort ( iar, \&lar ) ;
..
}
```

This will work, except maybe the name of the external sort is not correct (see External names, A.3.2).

A problem arises when passing character arrays to Fortran functions or subroutines. Most Fortran compilers expect somewhere the length of each string in it's arguments. Sometimes you must specify all the lengths at the end, sometimes immediately after each character argument. So if you have in your C-source two character arrays you want to pass to a Fortran subroutine it must look like this:

Fortran code: footnotesize

```
SUBROUTINE FORFUN (NAME, ADDR )
CHARACTER NAME*(*), ADDR*(*)
```

C-source: footnotesize

```
void forfun ( char *, char * ) ; /* prototype */
..
char name[25], address[40];
..
forfun (name, strlen(name), address, strlen(address) ;
..
    /* OR */
..
forfun (name, address, strlen(name), strlen(address) ;
..
```

Looking at the example above, it looks wise to define a macro for this function. The only machine dependent part of your source is in the macro. This macro could look like this:

footnotesize

```
#define FORFUN(p1, p2) \
            (void)forfun (p1, p2, strlen(p1), strlen(p2)

        OR

\#define FORFUN(p1, p2) \
            (void)forfun (p1, strlen(p1) , p2, strlen(p2)
```

and the call in the C-source is now:

footnotesize

```
FORFUN ( name, address );
```

If you put the Fortran prototypes and the macro's in a header file, all machine or compiler dependencies are located in one place. You can use pre-processor `#ifdef` statements to select the correct piece of code during compilation.

How to find out which way of argument passing you must use? Look in the C-manual of your compiler!!!!

A.3.2 External names

What name must you use in your C-code for the Fortran subroutine? This also differs from machine to machine. On most machines you can use the name of the Fortran function, in lower case. Some Fortran compilers add an _ to the begin or end of the name. You should look in the Fortran manual of the compiler you are using.

A.3.3 Return values

A Fortran subroutine does not return a value. A Fortran function however can return a value. This can be a long (`INTEGER*4`), short (`INTEGER*2`) or float (`REAL*4`) or double (`REAL*8`). For complex return-values, refer to the Manuals of your compiler.

Retrieving the return value from a Fortran character function is more complicated. The easiest way is not directly calling such a function from your C-code, but indirect using a Fortran subroutine, which calls the character function. This subroutine must also add the '
0' character to make it a C-string. The example below will make this clear:

The Fortran character function WORD: footnotesize

```
CHARACTER*(*) FUNCTION WORD ( VAR )
CHARACTER VAR*(*)
..
WORD = ......
..
END
```

The Fortran subroutine to call the above function: footnotesize

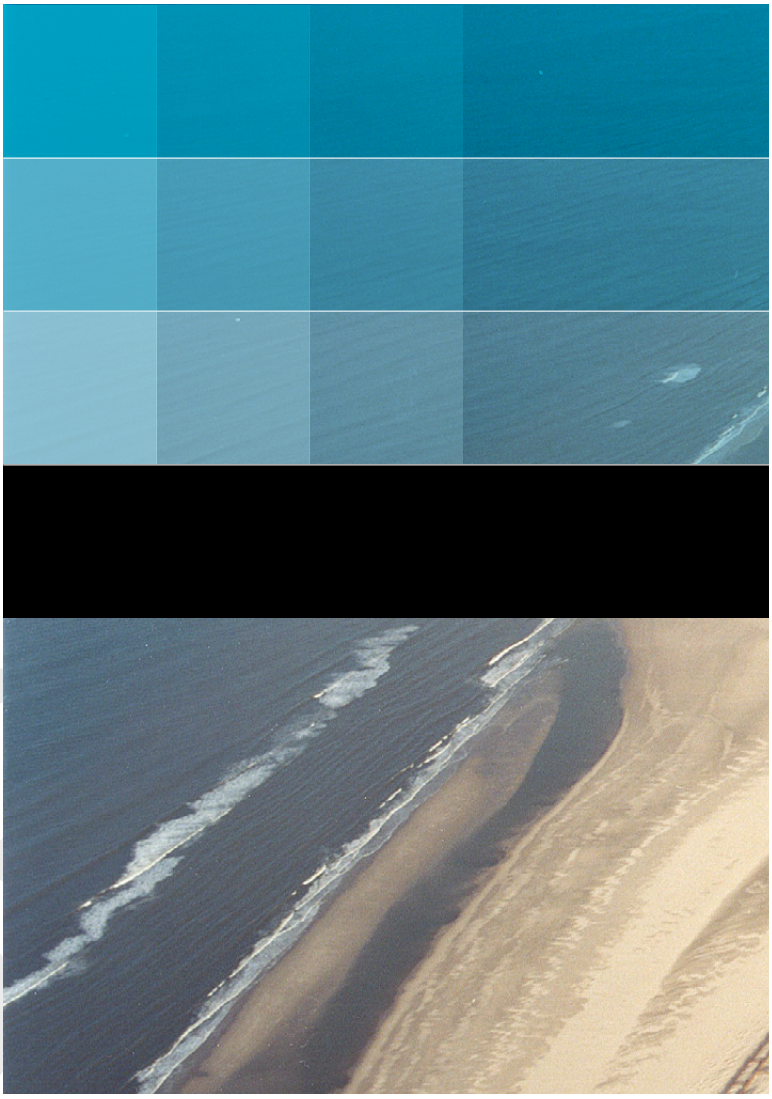
```
SUBROUTINE SBWORD ( VAR , RETVAL)
CHARACTER VAR*(*), RETVAL*(*)
CHARACTER WORD*20
..
RETVAL = WORD ( VAR )//CHAR(0)
..
END
```

Calling this function from your C-source: footnotesize

```
/* prototype for Fortran subroutine
   differs from machine to machine */
void sbword ( char *, char *, int , int );
..
main() {
..
    char retval[21];
   /* 1 character more than the function returns */
..
(void) sbword ( "input", retval, 5, 21 );
..
}
```

DRAFT

DRAFT



s systems

+31 (0)88 335 81 88
sales@deltaressystems.nl
www.deltaressystems.nl

DRAFT