C Reference Manual

ND-860251EN2

NorskData

C Reference Manual

ND-860251EN2

C Reference Manual

NOTE:
The numbering system for Norsk Data's documentation changed in September 1988. All numbers now start with an 8. The numbering structure is therefore ND-8xxxxx.xx xx. Example: ND-863018.3A EN. Existing manuals will receive a new number if and when they are updated or revised.

Version 1 March 1987
Version 2 January 1990

Norsk Data GmbH MANUAL

C REFERENCE MANUAL

Dokumentation ND Mülheim
29 November 1989

Norsk Data ND-860251.2 EN

Norsk Data ND-860251.2 EN

the product

Product Name	Product Number
C for ND-500	211149

The programming language C is one of today's most popular programming languages supporting many types of applications on a variety of different computers. C was originally designed by Brian W. Kernighan and Dennis M. Ritchie as the system programming language for the development of the UNIX® operating system. Nowadays, C has proven equally well suited for commercial, technical and scientific applications. The concept for modular programming, efficient language constructs and an extensive runtime system support are the keys to its success.

In order to provide a high degree of compatibility with other systems, the C language implemented by Norsk Data conforms to the specification described in "The C Programming Language" by Kernighan and Ritchie, which is generally regarded as C 'Standard'. The Norsk Data implementation offers useful language extensions and library support for many UNIX system functions as well as a good integration into the SINTRAN environment (symbolic debug and interfaces to SIBAS, ISAM, FOCUS and Monitor Calls).

the reader

This manual is intended for experienced programmers with or without knowledge of the C programming language.

prerequisite knowledge

The reader must have a basic knowledge of data processing and should also have some knowledge of the SINTRAN III operating system. Familiarity with C or a similar programming language, like PASCAL or PL/1, would be helpful.

® UNIX is a Trademark of Bell Laboratories.

the manual	In this manual you will find the description of the C language as it is implemented on the ND-500 machine. Furthermore, this manual contains descriptions of:

- C library functions including UNIX system calls,
- the interfaces to the Monitor Call, ISAM, SIBAS and FOCUS libraries.

related manuals

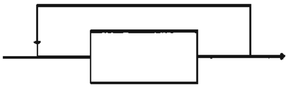
SINTRAN III Reference Manual...............ND-60.128 EN
SINTRAN III Time Sharing/Batch Guide.......ND-60.132 EN
SINTRAN III Monitor Calls..................ND-60.288 EN
SINTRAN III Real Time Loader...............ND-60.051 EN
PED Editor: PED User Guide.................ND-60.121 EN
ND-500 Loader/Monitor......................ND-60.136 EN
ND-500 Reference Manual....................ND-05.009 EN
Symbolic Debugger User Guide...............ND-60.158 EN
ND ISAM Reference Manual ....ND-60.108 EN
SIBAS II User Manual ....ND-60.127 EN
FOCUS Screen Handling System Ref. Manual...ND-60.137 EN
CAT-Profile................................ND-860307 EN

TABLE OF CONTENTS

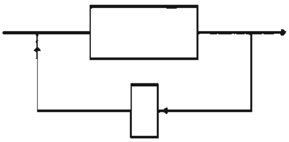
char	Keywords of the language C and compiler commands are printed in **bold** letters.
@	This is the SINTRAN III prompt sign. It indicates that you are in connection with the operating system and can enter SINTRAN commands.
@list-files	Text to be typed in by the user is underlined. This applies especially to compiler commands or SINTRAN commands. (Program code, which can be typed in by the user as well, is not underlined.)
	The following notation is used when describing compiler commands:
help <command : >	Required parameters are included in angle brackets.
page-length [<lines: >]	Angle brackets enclosed by square brackets indicate optional parameters. They can only be specified in the command line and will not be prompted for in the dialogue.
options <option: >...	If more than one value may be specified, the right bracket will be followed by three dots.

syntax diagrams

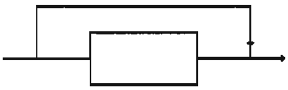
Throughout this manual the following structures will be used to describe the syntax of C:



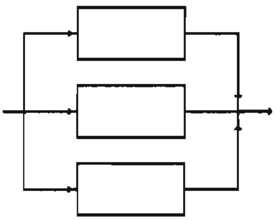
This structure indicates that the construct in the rectangle may occur an indefinite number of times, but at least once (iteration).



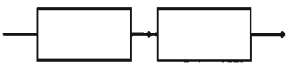
The construct in the bigger rectangle is separated by those (comma, semicolon, etc) in the smaller rectangle.



By-passing arrows mean that the construct in the rectangle may be skipped (option).



This structure describes alternatives.



This structure is a sequence of constructs.

Chapter 1

Introduction

A General Overview

C is a compact programming language with few but powerful basic constructs.

control structures

Like PASCAL, C provides the fundamental sequencing statements:

- compound statements ({})
- decisions (**if**)
- loops (**while**, **for**, **do**)
- selection of a case out of a range of alternatives (**switch**)

functions

C programs can be split up into several functions, which may be compiled separately, thus allowing modular programming. Nesting of functions is not allowed.

input/output

The I/O-system is not a part of the C language, i.e. there are no special statements for reading and writing. Input and output operations can be effected by calling functions of the C library.

operators

Significant for C is its wide range of operators allowing powerful expressions in arbitrary context.

general characteristics

In addition to the common constructs in higher programming languages, C offers the following features:

- complex initialisations at compile time
- constant expressions within declarations
- declarations in local blocks
- function variables
- functions returning structures
- assignment operations within expressions
- operations for bit manipulations
- pointer arithmetic
- comfortable operations on files
- include files and include hierarchies
- conditional compilation
- optional runtime checks (pointer check, index check, subrange check)

A Simple C Program

In this section a small C program is introduced, showing the basic elements of programming in C. The program produces a table, which lists the frequency of the different input letters. If you want to try it out:

- enter the code in PED and store it (see PED Editor)
- enter the C compiler with the SINTRAN-command

  @NC ↵

- compile and link your program with the command

  NC: link <source file: >,<program: > ↵

- start it from SINTRAN by entering *ND* and the program name

  @ND program-name ↵

example

```
include <stdio.h>
int COUNT[26];
main ()
{
   int CH;
   printf ("Please type your text\n");
   printf ("Terminate your input by CR\n");
   CH = getchar();                         /* read input character    */
   while (CH != '\n')                      /* '\n' gives the value of */
      {                                    /* the newline character   */
      CH = lower (CH);                     /* call function lower     */
      if ('a' <= CH  &&  CH <= 'z')
         ++COUNT[CH - 'a'];                /* count letters           */
      CH = getchar();
      }
   printf ("Frequency of letters:\n");     /* print title             */
   for (CH= 0;  CH < 26;  ++CH)            /* print listing           */
      if (COUNT[CH] != 0)
        printf ("    %c : %d\n", CH + 'a', COUNT[CH]);
}


lower (CH)                  /* function to convert upper into lower  */
   int CH;                  /* case letters                          */
   {
   if ('A' <= CH  &&  CH <= 'Z')
      CH += 'a' - 'A';
   return (CH);
   }
```

#include *<stdio.h>*	With the first statement the file *stdio.h* is included, which contains the declarations of I/O functions like *getchar*.
int *COUNT[26]*;	This statement defines an integer array with 26 elements. Array indices always start at zero, i.e. *COUNT* has the elements *COUNT[0]* up to *COUNT[25]*. The semicolon defines the end of the statement. As this declaration is written outside all blocks it is a global declaration, and the array elements will be initialised to zero automatically.
the function *main*	Execution of a C program always starts with a function called *main*. Parameters can be passed within round brackets. In our example, *main* has no parameters, but for syntactical reasons the brackets still have to be written: *main()*.
a block: {...}	The curled brackets { and } combine all included statements to <u>one</u> compound statement or block. They can be compared to the DO-END block in PL/1 or the BEGIN-END statement in PASCAL.
int *CH*	This statement defines an integer variable, which is local to *main*. It will not be initialised automatically.
printf	The library function *printf* sends a message to the terminal. For more details see below.
getchar	Each call of the I/O function *getchar* returns the next input character. The default input device is the terminal.
/* text */	Any text enclosed by /* and */ is interpreted as comment and will be ignored by the compiler.
while (CH != '\n');	In the statements controlled by this **while** condition the function *lower* is called, letters are counted and the next input character is read. These statements are surrounded by curled brackets, which combine them to one compound statement. They will be executed as long as there is input. The operator != means not equal and *\n* is the newline character. As it is surrounded by apostrophes its numerical value will be taken for the comparison (see also **character *constant*** on the next page).

the function *lower* — The function lower is called with the current input character as parameter. In order to be known to the function, the declaration of the parameter must appear after the parameter list and before the curled left bracket at the beginning of the function definition. The condition of the **if** statement tests whether the input character is an upper case letter. The operator && represents logical AND.

CH += 'a' - 'A' — This statement serves to convert an upper case letter into a lower case letter. It could also be written as

CH = CH + ('a' - 'A')

Any single character can be written between apostrophes to produce a value equal to the numerical value of the character in the machine's character set. This is called a 'character constant'. In the ASCII character set 'A' has the value 65 and 'a' the value 97 (see page 16-77).

Assuming the input character was *'B'*, then *CH* gets the value 98.

++COUNT[CH-'a'] — Here the operator ++ increments an array element by one. This statement could also be written as

COUNT[CH-'a'] + 1

The expression of the array index *CH-'a'* reduces the value set of the array indices to 0-25. If the input character was *'B'* and *CH* now consequently has the value 98, the value of the array index is 1.

for — A for statement has three parts separated by semicolons.

The first part *CH = 0*;

initialises the control variable. It is only executed once, and well at the beginning of the loop.

The second part *CH < 26*;

is the condition controlling the loop. As long as the condition is true, the loop will be executed.

The third part *++CH*;

increments the control variable by 1 every time the loop is repeated.

printf

printf is a library function with the terminal as default output device. It is a format conversion function for general purposes. In our example, the first call of *printf* sends the title *Frequency of letters:* to the terminal.

```
printf ("Frequency of letters:\n");
```

is equivalent to

```
printf ("Frequency ");
printf ("of ");
printf ("letters:\n");
```

newline : \n

\n is an escape sequence representing a newline character, which produces a carriage return and a line feed in the output.

The first parameter in the statement

printf ("   %c : %d\n", CH + 'a', COUNT[CH])

%

is a string of characters to be printed, with each **%** sign indicating <u>where</u> the following parameters are to be substituted, and <u>what form</u> they are to be printed in. Each **%** construction in the format string is paired with one of the following parameters. Generally, the number of **%** constructions should correspond to the number of remaining parameters.

%c - character

In our example, *%c* means that a character is expected as corresponding parameter *(CH + 'a')*. It has to be printed on the fourth position, followed by a blank, a colon and another blank.

%d - decimal integer

%d means that a decimal integer is expected as corresponding parameter *(COUNT[CH])*.

The escape sequence *\n* specifies that each line has to end with a carriage return and a line feed.

According to the above rules, the first three output lines of our example program could look like this:

```
Frequency of letters:
   a : 16
   b : 4
```

Chapter 2

Basic Elements

Character Set

All lexical symbols, e.g. identifiers, keywords, etc., are built from the characters of the ASCII character set (see page 16-77). Alternative characters are separated by a |.

letters

A | B | C | D | E | F | G | H | I | J | K | L | M |
N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
a | b | c | d | e | f | g | h | i | j | k | l | m |
n | o | p | q | r | s | t | u | v | w | x | y | z

octal digits

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

decimal digits

octal digit | 8 | 9

hexadecimal digits

decimal digit | a | b | c | d | e | f | A | B | C | D |
E | F

special signs

+ | - | * | / | = | < | > | [|] | (|) | { | } | . |
, | : | ; | ^ | ' | <blank> | ! | " | # | $ | % | & |
\ | _

Keywords

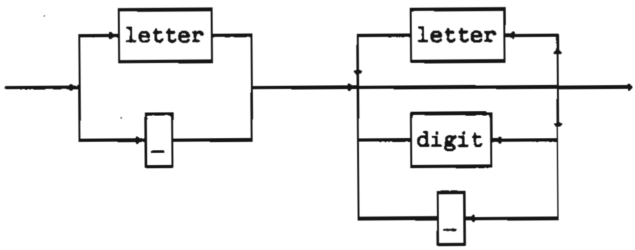
Keywords are pre-defined character strings. They are reserved for certain purposes, i.e. they must not be used as identifiers (see "Identifiers" below). In C, keywords are written in lower case letters. The following keywords are defined:

auto	**do**	**float**	**register**	**switch**
break	**double**	**for**	**return**	**typedef**
case	**else**	**goto**	**short**	**union**
char	**entry**	**if**	**sizeof**	**unsigned**
continue	**enum**	**int**	**static**	**void**
default	**extern**	**long**	**struct**	**while**

Identifiers

An identifier (ID) is a name that designates a data element, like a constant, a type, a variable or a function.

IDENTIFIER (ID):



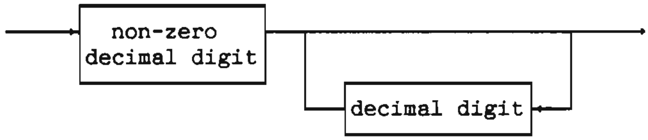
An identifier is a sequence of letters, digits and/or underscore, starting with a letter or underscore sign. It may consist of at most 32 characters all of which are significant for the compiler and the ND-500 linkage loader.

Distinction is made between upper and lower case letters, i.e. *name_1* and *NAME_1* are two different identifiers.

Constants

Before we can define the different kinds of constants, we first have to define some basic elements:

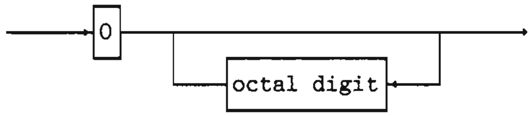
DECIMAL NUMBER:



In order to distinguish between decimal and octal numbers, a decimal number must always start with a non-zero decimal digit.

example 123

OCTAL NUMBER:

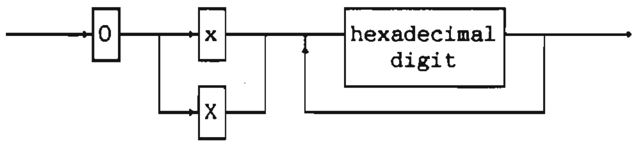


An octal number always starts with a 0 (zero).

The decimal digits 8 and 9 have the octal values 10 and 11 respectively.

example 0177

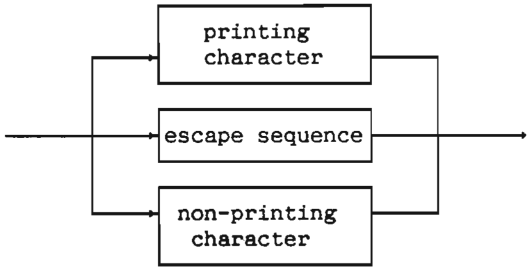
HEXADECIMAL NUMBER:



A hexadecimal number always starts with the sequence 0x or 0X (digit zero and letter x). The letters a..f or A..F represent the decimal values 10 to 15.

example 0x7FFFF

CHARACTER LITERAL:



printing character

A printing character is a character of the ASCII character set as described in the section "Character set" on page 2-3.

escape sequence

An escape sequence is a mechanism to represent control characters. It is written as a backslash followed by a character. But, although it is written as two characters, it is stored as one single character. Below the available escape sequences are listed together with their meanings:

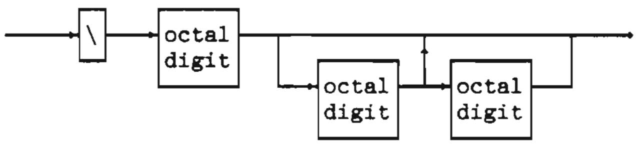
\b	backspace
\n	newline
\r	carriage return
\t	horizontal tab
\f	form feed
\\<blank>	blank
\\\\	backslash
\\"	quotes
\\'	apostrophe

non-printing character

null character \0

A non-printing character has an ASCII value smaller than 32 decimal. It is represented by a backslash followed by 1, 2 or 3 octal digits which specify the numeric value of a bit mask. A special case of this construction is \0, which indicates the null character, whose value is zero. \0 is the terminating character for a string.

NON-PRINTING CHARACTER:



example

'\15' == '\015' == 0xD == 015 == 13 (=='\n' (SINTRAN only)).

Now we can define the constants:

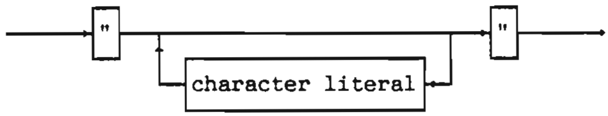
CHARACTER CONSTANT:

The value of a character constant is the numerical value of the character literal in the machine's character set. Character constants can be used in numeric operations just as ordinary numbers. Most often they are used in comparisons with other characters. Our introduction example on page 1-4 shows an application possibility. The character values are listed in Appendix A on page 16-77.

example

'a' = 97

STRING CONSTANT:



A string constant is a sequence of characters surrounded by quotes. It is a character array and has storage class *static* (see pages 3-6 and 4-3). The compiler automatically places a null byte (\0) at the end of each string to mark it for scans.

storage required

Therefore, the storage required is one byte more than characters between the quotes. The same escape sequences as in character constants can be used, e.g. a double quote within a string must be preceded by a backslash (\").

maximum length

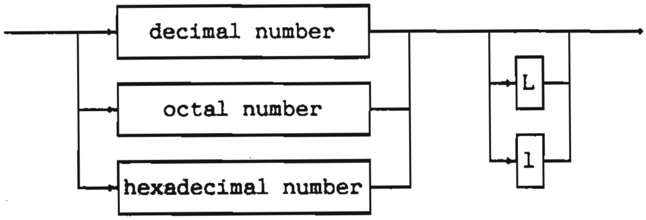
The maximum length for a string constant is 4096 characters including the quotes and escape sequences. A string may be continued on the next editor line by placing a backslash (\) immediately followed by CR at the end of the line to be continued.

example

" \"HALLO\" "

This string constant requires 10 bytes storage (including the terminating null character \0).

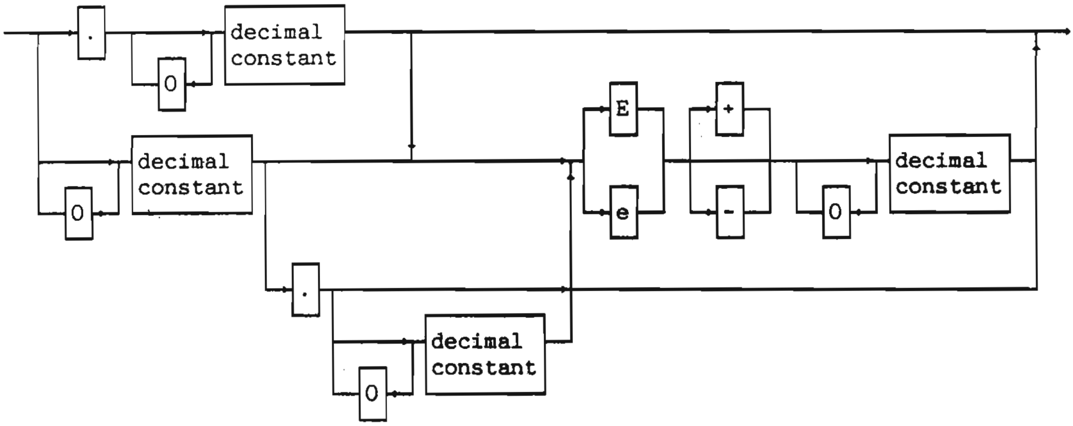
INTEGER CONSTANT:



long — An integer constant followed by the letter L (or lower case l) is stored as a long constant.

short — Short decimal integer constants are implicitly taken to be long if their values exceed 32767; short hexadecimal and octal constants become long if their values exceed 65535.

FLOATING CONSTANT:



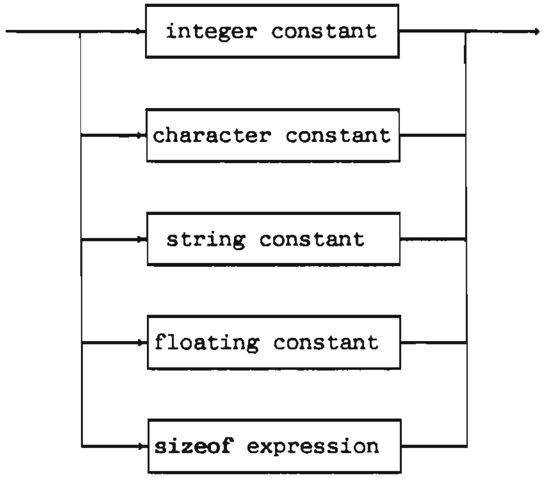
A floating constant consists of an integer part, a decimal point, a fraction part, an e or E, and an optionally signed integer exponent. Either the integer part or the fraction part (not both) may be missing, and either the decimal point or the E and the exponent (not both) may be missing.

Every floating constant is taken to have double precision.

examples — 0.4 | 5e10 | 77. | .10e-1 | 6.54321E+8

There are five different kinds of constants:

CONSTANT:



A **sizeof** expression will be explained on page 7-9.

Comments

Comments are included by the symbols /* and */. They may appear anywhere where blanks or new lines are allowed and are ignored by the compiler.

example

```
/* Comments should be used    */
/* for program documentation. */
```

Chapter 3

Data Types

Simple Types

The main simple types are **char**, **int** and **float**. Examples for these data types are the character, integer and floating constants on page 2-7.

char

A data element of type **char** can contain any member of the character set. Its value is equivalent to the integer code for that character (see "ASCII Character Set" on page 16-77). The data type **char** has a range of -128.. +127, while the range of **unsigned char** is 0..255.

int

The keywords **short int**, **int** and **long int** describe three integer subranges. Furthermore, integers can be qualified as **unsigned**. The range of unsigned integers is defined by arithmetic

modulo 2^n

where *n* is the number of bits used to store the integer (see tables below). Unsigned integers are always positive.

float

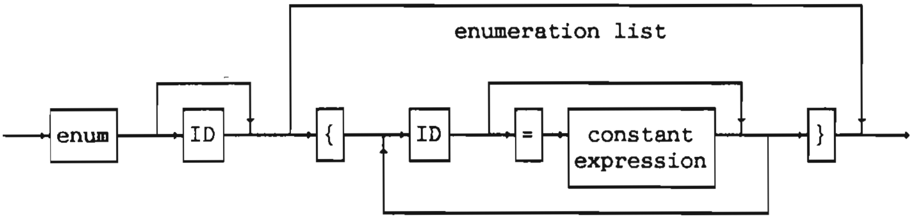
The type **float** describes a single precision floating point value, whereas **long float** or **double** describe a double precision floating point value.

SIZE, RANGE AND PRECISION OF SIMPLE TYPES

TYPE	SIZE	RANGE	PRECISION
char	1 byte	-128..+127	-
unsigned char	1 byte	0..255	-
short int	2 bytes	-32768..+32767	-
unsigned short int	2 bytes	0..65535	-
(long) int	4 bytes	-2147483648..+2147483647	-
unsigned (long) int	4 bytes	0..+4294967293	-
float	4 bytes	$\pm 10^{76}$	7 digits
double	8 bytes	$\pm 10^{76}$	16 digits

enumeration types — Enumeration types are used to associate names with integer constants.

ENUMERATION SPECIFICATION:



enumeration identifier — The first identifier is the name of the enumeration. The use of enumeration identifiers is described on page 4-6. The enumeration identifier and all identifiers of the enumeration list must be disjunct.

enumeration list — The identifiers of the enumeration list can be used wherever integer constants are allowed. If no constant expression is specified the first identifier of the list will be represented as 0(zero), the second one as 1, etc. incrementing by 1 from left to right. Alternatively, you can explicitly assign an integer value to an identifier; the following identifiers will get incremented values based on this assignment.

constant expression — The constant expression must evaluate to a value of type **int**. The exact definition of a constant expression is given on page 7-15.

Each enumeration is regarded as an individual type.

examples

- **enum** SPEC1
- **enum** TREE {OAK,MAPLE,BEECH}

  where OAK=0, MAPLE=1, BEECH=2
- **enum** {red=0, green=5, yellow, blue}

  where red=0, green=5, yellow=6, blue=7

Implicit Type Conversions

Type Conversions

When combining values of different types in an arithmetic operation, type conversions will be done implicitly according to the following rules:

1. First, any operands of type **char** or **short int** are converted to **int**, and any of type **float** are converted to **double**.

2. Then, if either operand is **double**, the other is converted to **double**, and that is the type of the result.

3. Otherwise, if either operand is **long int**, the other is converted to **long int**, and that is the type of the result.

4. Otherwise, if either operand is **unsigned**, the other is converted to **unsigned**, and that is the type of the result.

5. Otherwise, both operands must be **int**, and that is the type of the result.

conversions by assignment

For the conversion of a **long** integer value into a **short** integer or **char** value the most significant bits will be truncated.

When assigning a **double** value to a **float** value the mantissa will be rounded and then truncated.

For the conversion of **float** into integer the decimals will be truncated. If the **float** value is too big to be represented as an integer, the result is undefined.

explicit conversion

Explicit type conversions are described on page 3-16.

Composed Types and Pointers

As composed types and pointers may be rather complex, we first describe the main, and simpler, aspects of these types. This will enable you to understand better the syntax diagrams of type definitions and specifications at the end of this chapter. There you also will find some examples of more complex type definitions. In the chapters "Pointers and Arrays" and "Functions" applications and special details of these types will be described.

Arrays

In C, an array is declared by an identifier followed by the sizes of the dimensions in square brackets. All elements of an array have the same specified type.

subscripts

An array subscript may be any positive integer expression, e.g. an integer variable or constant. Array subscripts always start at 0.

examples

int DIGIT[10]

/* *DIGIT* is an array with 10 integer elements: */
/* *DIGIT[0]* up to *DIGIT[9]* */

char ABC [10] [2] [5]

/* *ABC* is a three-dimensional array with 100 */
/* character elements. In other words, *ABC* is */
/* an array with 10 items; each item is an */
/* array of 2 arrays; each of the latter arrays */
/* is an array of 5 characters. */

> NOTE
> An element of an array must not be a function (see page 3-9); only pointers to functions are allowed.

Structures

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. In some other languages, e.g. PASCAL, structures are called records.

struct

A structure specification starts with the keyword **struct** followed by an optional structure identifier and, in braces, a list of component declarations. Each structure specification describes an individual type, i.e. two different structures describe two different data types.

example

```
struct EMPLOYEE {
       char name[16];
       int telephone[3];
       int department;
       }
```

Three operations on structures are allowed:

- The address of the structure can be determined (see page 5-5).
- A structure may be assigned to a structure variable of the same type or may be returned as the result of a function.
- Components can be accessed. A component of a structure can be handled like any other variable. One method of referencing a component is:

component reference

```
structure_variable_id . component_id
```

Components can also be referenced by using pointer arithmetic (see page 5-5).

example

```
struct {
       float COMP1;
       char COMP2;
       } ST_VAR;
```

ST_VAR is a structure variable identifier. The components are referenced as

```
         ST_VAR.COMP1
   and   ST_VAR.COMP2
```

Unions

size of a union

Unions allow the user to have overlapping data in a single area of storage. Its main purpose is to save storage. Their data structure corresponds to the variant record (CASE) of PASCAL. A union is a variable that may contain objects of different data types and sizes, but only one at a time. It has the size of its largest component.

union

The syntax of a union specification is identical to a structure specification, except for the keyword. The keyword **struct** has to be replaced by the keyword **union**.

examples

```
union UNION_ID {
      int INTVALUE;
      char CHARVALUE;
      short SHORTARRAY[3];
      }
```

SHORTARRAY is the largest component of above union. It requires 6 bytes storage (a **short** value is stored in 2 bytes). Thus, the size of above union is 6 bytes.

A union can also be a component of a structure:

```
struct d {
       int day;
       union {
             int mon_nr;
             char mon_name[4];
             } month;
       int year;
       }
```

current contents

At runtime, the actual type and value of a union is determined by the last assigned variable. It is the programmer's responsibility to keep track of the data type that is currently stored in a union.

Note

A component of a structure or union must not be a a function; only pointers to functions are allowed. (See sections "Pointers" and "Functions" on the next page.)

Pointers

A variable of type pointer contains the address of another variable. It is defined by an asterisk (*) directly followed by a variable identifier. In the following declaration **V* is a variable of type *T* and *V* is a pointer containing the address of **V*:

*T *V*

> **Note**
> A pointer can only refer to an object of the defined type. A pointer declared as **int** **V* can only point to an object of type **int**.

examples

```
int *a [5];   /* array of 5 pointers to integer    */
int (*a) [5]; /* pointer to an array of 5 integers */
```

Functions

Functions are sequences of declarations and statements that can be called by their name and may return a result value. Functions are used to

- structure a program into logical units,
- save storage and double writing, and
- save programming effort. Typical problems can be solved in functions, which can also be used by other programs.

A simple function definition consists of

- an optional type specifier (e.g. **int**) to specify the type of the result value,
- a function identifier, and
- an optional parameter list in round brackets.

As result type for functions the special type **void** can be specified, which indicates that the function does not deliver a result value. If no result type is specified **int** is the default type.

A function is activated by the appearance of its identifier. Functions returning a value may be part of an expression. Calls of functions without a return value are statements.

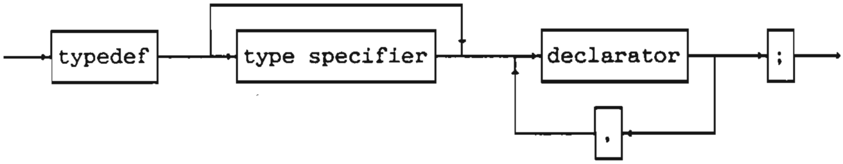
examples

```
int F1()    or    F1()

/* F1 is a function without parameters.     */
/* It returns a value of type int.          */

char *F2(P1,P2)

/* The function F2 has two parameters.     */
/* It delivers a pointer to a char object */

struct {
        int s1;
        long s2;
       } F3 ()

/* The function F3 has a structure as      */
/* result type                             */

(*F4)()

/* The function F4 delivers a pointer      */
/* to a function of type int               */
```

Type Definitions

Type definitions do not reserve storage. As in PASCAL they define type identifiers which denote the type specified and can be used in other type definitions or declarations.

TYPE DEFINITION:



The following syntax diagrams give an exact definition of a type specifier and a declarator. Both the type specifier and the declarator are also elements of other syntax diagrams. Therefore, we advise you to study them carefully.

TYPE SPECIFIER:

unsigned

unsigned char

unsigned short

default type: **int**

short int

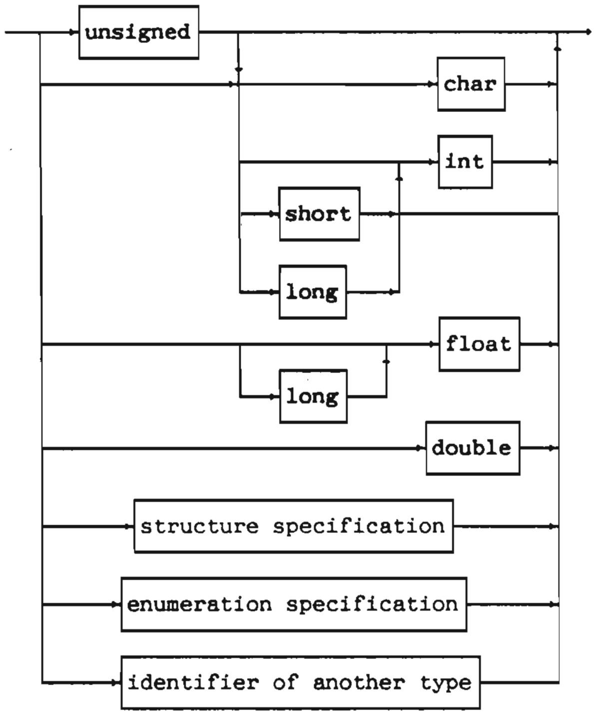
long int

float

long float = double

(see page 3-13)

(see page 3-4)



If no type is specified **int** is the default.

DECLARATOR:

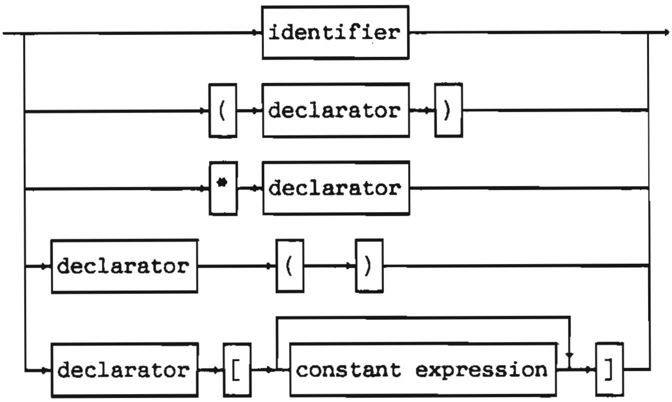
simple variable

brackets change priority of operators

pointer

function

array



Examples for constant expressions are integer and character constants. The exact definition of a constant expression is given on page 7-15. The priority of operators is described on page 7-10.

examples

```
typedef enum {red, blue, green} COLOUR;

typedef COLOUR C1, C2;
```

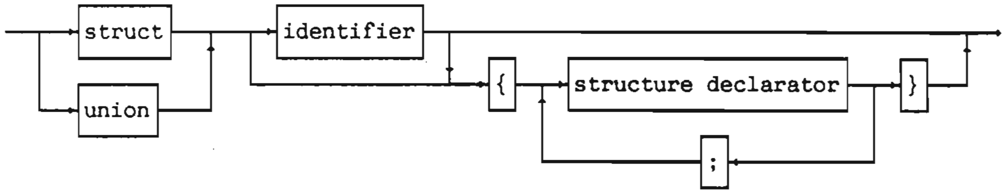
type identifiers

```
/* The type identifiers COLOUR, C1 and C2 */
/* denote the same type, i.e. the above   */
/* defined enumeration.                   */


typedef unsigned *F1(), (*P1)();

/* Brackets in a declarator :                */
/* F1 is a function that returns a pointer   */
/* to an unsigned integer object. P1 is a    */
/* pointer pointing to a function that       */
/* returns an unsigned integer. Syntacti-    */
/* cally, F1 and P1 are type identifiers.    */
```

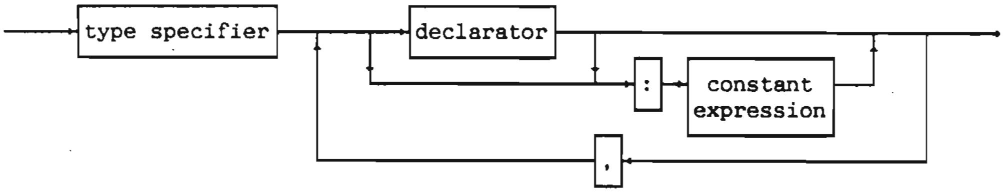
Note

Type definitions must not be repeated!

STRUCTURE SPECIFICATION:



STRUCTURE DECLARATOR:



examples

```
typedef struct S {
            double (*COMP1)();
            unsigned COMP2, COMP3;
            short COMP4;
            } ST;
```

structure identifier
type identifier

```
/* The structure identifier of above structure   */
/* is S, whereas the type identifier is ST. The  */
/* first component COMP1 is a pointer to a       */
/* function of type double. The second and third */
/* components are defined as unsigned integers   */
/* and the fourth one as a short integer.        */
```

```
typedef struct S2 {
            char [20] [10] STRINGS;
            union {
                  char LETTER;
                  int CODE;
                   } NC;
             } T2;
```

structure identifier

type identifier

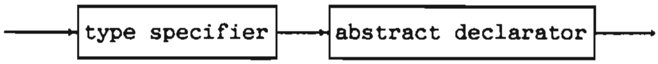
```
/* S2 is the structure identifier, describing      */
/* everything between the braces, while T2 is a    */
/* type identifier. The component STRINGS speci-   */
/* fies an array of 20 strings with a length of    */
/* 10 characters. The second component NC con-     */
/* tains either an character or an integer.        */
```

Type Names

Type names are needed for explicit type conversions (see next page) and in **sizeof** expressions (see page 7-9).

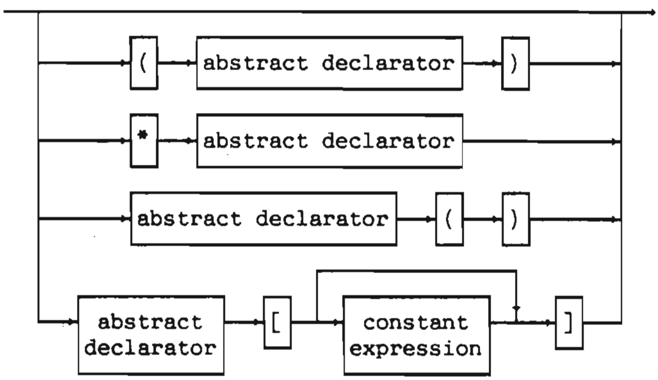
Be careful not to confuse type names and type identifiers (see page 3-11-3-13).

TYPE NAME:



where an abstract declarator is defined as:

ABSTRACT DECLARATOR:



A constant expression can be an integer, a character or an enumeration constant. At compile time it evaluates to an integer constant.

examples

```
char *
double ()
int [5]
float
```

Explicit Type Conversions

Explicit Type Conversions

You can force an explicit type conversion by using the cast construct:

```
(type name) expression
```

The value of the expression is converted into the specified type according to the rules of implicit conversion (see page 3-5). Generally, the expression is a variable identifier.

pointer conversion

As pointers cannot be converted implicitly, the cast construct is mainly used for the conversion of pointers.

examples

```
int NUMBER;      /* NUMBER is declared as integer */
float F;         /* F is declared as float        */
char *P;         /* P is a pointer to char        */
double *D;       /* D is a pointer to double      */


   F = (float) NUMBER;
   D = (double *) P;

/* Before the assignment NUMBER is converted  */
/* into float and P into a pointer to double. */
```

parameter passing

Another important application of cast constructs is the conversion of function parameters to be passed:

example

```
   double R, sin();
   int I;

/* We assume, that the function sin requires a */
/* parameter of type double:                   */

   R = sin((double)I);

/* With the cast construct I is converted into  */
/* double before being passed to sin.           */
```

Chapter 4

Declaration and Initialisation of Variables

Storage Classes

The storage class of an identifier determines its scope and its lifetime.

local

A variable declared within a block or function is local to its block, i.e. its scope is restricted to the block where it is declared. Local variables cannot be referred to from outside the block.

global

A variable declared outside all blocks, i.e. outside the function *main*, has a global scope. Its scope is either the entire program or restricted to its source file. As functions cannot be nested they always have a global scope.

In C, there are four storage classes:

automatic (**auto**)
external
static
register

auto

- Only local variables can be declared automatic. Each time when entering a block, their value is undefined (other than local **static** variables - see below).

external

- External variables have a global scope. They exist and retain their values throughout the execution of the entire program. External variables can be used to communicate between functions or separately compiled modules. Thus, they offer an alternative to the data passing of function parameters and result values, especially when long parameter lists are needed. However, you should be careful with external variables, as they can be altered by different functions.

local **static**

- A variable declared **static** within a function has a local scope like automatic variables. The only difference is that **static** variables retain their values until the next invocation of the block.

global **static**

A variable declared **static** outside all functions has a global scope like external variables, with the restriction to the source file. Functions compiled in a different file have no access to that variable.

register

- Variables of storage class **register** behave like automatic variables. If the compiler performs register optimisation, they are stored in the hardware registers of the machine, which leads to a faster execution.

default storage class

If no storage class is specified the default storage class for a variable declared within a block is **auto**, and for a variable declared outside all blocks external.

example

```
                /* SOURCE FILE 1 */

/* Any external declaration here (outside all func-*/
/* tions and without the keyword static) defines a */
/* global scope throughout the entire program for  */
/* that variable, i.e. such a variable can be ac-  */
/* cessed in source file 1 and source file 2.      */

  main()
  { /* begin main */

    /* Any variable declared here can be accessed*/
    /* in block 1 or 2, but not in function_1.   */

    { /* begin block 1 */

      /* Any declaration inside this block is */
      /* local to block 1. A declaration of   */
      /* the samme variable identifier outside*/
      /* this block becomes invalid inside.   */

    } /* end block 1   */

    {
       /* block 2      */
    }

  } /* end main   */
```

```
                /* SOURCE FILE 2 */

/* Any global declaration in this place including  */
/* the keyword static is only global within this   */
/* source file.                                     */

   function_1()
   {
     ...
   }
```

Declarations

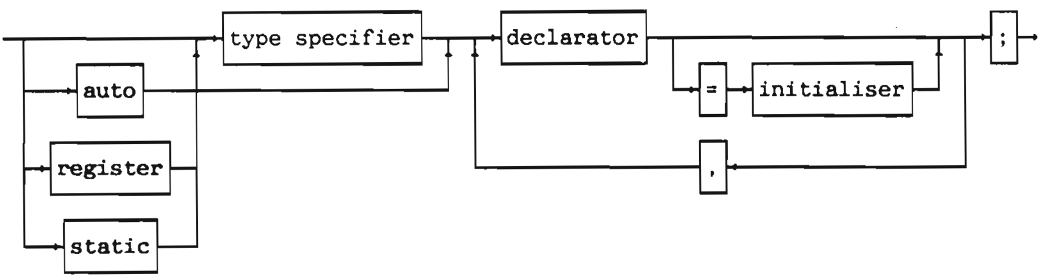
Declarations

Except integer functions, all variables and functions must be declared before use. A declaration specifies either a storage class or a type or both, and is followed by one or more variable identifiers. Furthermore, the declaration may also include an initialisation.

implicit function declaration

An unknown identifier followed by a left bracket is implicitly declared as a function with **int** as result type.

DECLARATION:



The initialiser is described on page 4-9.

defaults

REMEMBER: If no storage class is specified the default storage class for a variable declared within a block is **auto**, and for a variable declared outside all blocks external. The default data type is **int**.

> Note
> A function may only be declared external or **static**.

redeclarations

Generally, redeclarations of variables and functions should be avoided. A redeclaration with identical type and storage class will be ignored; if you redeclare an external variable or function as **static**, this redeclaration will be ignored as well. A redeclaration of a structure or union will lead to an error.

example

```
static Y;
{
    double X;
  ...
}

/* As X is declared inside a block, it implicitly */
/* gets storage class auto. Y is declared outside */
/* all blocks, i.e. it is a global variable.      */
/* Its type is int.                               */
```

identifiers

In type definitions of structures, unions and enumerations two kinds of identifiers may be defined:

- The first one before the braces is optional. It designates the kind of structure, union or enumeration.
- The second one behind the braces is the type identifier, which is mandatory.

Both identifiers may be used in declarations:

structure declaration

struct STRUCTURE_ID VARIABLE_ID ;
is equivalent to
STRUCTURE_TYPE_ID VARIABLE_ID ;

union declaration

union UNION_ID VARIABLE_ID ;
is equivalent to
UNION_TYPE_ID VARIABLE_ID ;

enumeration declaration

enum ENUMERATION_ID VARIABLE_ID ;
is equivalent to
ENUMERATION_TYPE_ID VARIABLE_ID ;

example

typedef struct STR_ID {...} STYPE_ID;

Given the above type definition, there are two alternative ways to identify a structure in a variable declaration; you can use the structure identifier as in:

STYPE_ID VAR_ID;

or the type identifier as in:

struct STR_ID VAR_ID;

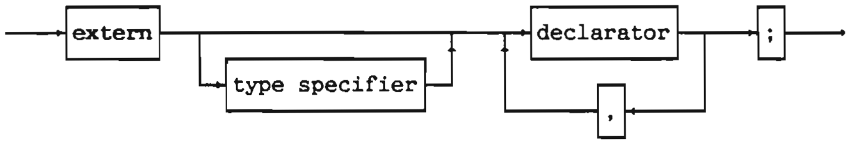
Both declarations are equivalent.

Extern Specification

Extern Specification

If you want to access (i.e. import) variables or functions that are declared in other source files, you have to specify them as **extern**.

EXTERN SPECIFICATION:



type specifier

For clarity's sake, an **extern** specification should always contain a type specifier; it must be the same type specifier as in the declaration of the source file.

> Note
> You can only import external variables or functions, i.e. global variables or functions that are not declared as **static**.

example

source file 1:

```
float ITEM;

main()
{
  ...
  ITEM = 5;
  F2();
  ...
}
```

source file 2:

```
extern float ITEM;

F2()
{
  ...
}
```

Initialisations

Initialisations For initialisations the following rules generally apply:

implicit initialisation

- External and **static** variables are only initialised once, and well at the beginning of program execution. If there is no initial value specified in the declaration, they are implicitly initialised to zero.

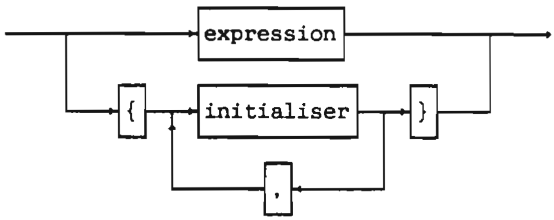
explicit initialisation

- Automatic and **register** variables must be initialised explicitly before use, otherwise their contents is undefined.

initialiser

- For external and **static** variables the initialiser is restricted to being a constant expression; automatic or **register** variables may be initialised by any expression involving previously defined values. Even a function call as part of the expression is allowed.

INITIALISER:



examples

```
float FL;              /* FL and L are implicitly    */
long L;                /* initialised to zero. */
func()
{
 int N = 2;           /* N, X, Z and A are initia-  */
 int X = N - 1;       /* lised each time the block  */
 float Y,Z = 0;       /* is entered. Y is undefined.*/
 char A = '\0';
  ...
}
```

Note

A union cannot be initialised.

Initialisation of Arrays

Arrays can be initialised by a list of initialisers enclosed in braces and separated by commas.

examples

```
short digit[10] = {0,1,2,3,4,5,6,7,8,9};

int number[8] = {20,15,10,5};

char word1[] = {'w','o','r','d','\0'};

char word2[5] = "word";

char hex[] = {'a','b','c','d','e','f'};

  {
    ...
  }

/* The above arrays have 10, 8, 5, 5 and */
/* 6 elements.                           */
```

If there are fewer initialisers than the size of the array, the other elements will be set to zero. It is not possible to specify repetition of one initialiser and an element in the middle of the array can only be initialised by giving the preceding values as well.

size calculation

If the size of an array is not specified the compiler calculates it by counting the number of initialisations. Because of the null character the size of a character array is always one more than the length of the string constant.

character arrays

As you can see in above examples, character arrays can be initialised in two ways:

- by specifying single values and the null character at the end, or
- by giving a string.

multi-dimensional arrays

Each row of a multi-dimensional array is initialised like a one-dimensional array. As the compiler can only calculate the size of the first dimension, the other dimension sizes have to be specified by the user explicitly.

examples

```
static int days[2][12] = {
        {31,28,31,30,31,30,31,31,30,31,30,31},
        {31,29,31,30,31,30,31,31,30,31,30,31} };

char MONTH[][10] = {
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December" };
{
  ...
}

/* The array MONTH has twelve lines with a       */
/* maximum of ten characters each ("September" */
/* is nine characters long plus the null char- */
/* acter). The size will be calculated as       */
/* 12 * 10 * 1 bytes. The lines with the shor-  */
/* ter month names will be padded with null     */
/* characters.                                  */
```

Initialisation of Structures

You may only initialise external or **static** structures.

The initialiser is specified in the same way as for one-dimensional arrays, i.e. the values are enclosed in braces and separated by commas.

examples

```
static struct date {
            int day;
            struct {
                  int month_nr;
                  char month_name[4];
                  }
            int year;
            } = {4,{3,"APR"},1987};
```

When the initialisation list is complete, the inner braces may be omitted.

examples

```
struct {
       char *day;
       int number;
       } count[] =
            { "Monday",0,
              "Tuesday",1,
              "Wednesday",2,
              "Thursday",3,
              "Friday",4,    };
```

Initialisation of Pointers

There are only two meaningful initialisers for a pointer:

- the value 0 (zero), which indicates that the pointer is pointing nowhere, or
- an expression involving addresses of previously defined data of appropriate type.

examples

```
int DIGIT;
int *A = 0;              /* A is pointing nowhere */
int *NUMBER = &DIGIT;

   /* NUMBER is declared as an integer pointer   */
   /* initially containing the address of DIGIT. */
```

Chapter 5

Arrays and Pointers

Relationship Arrays - Pointers

There is a close relationship between arrays and pointers. In fact, each array identifier is a pointer to the first element of an array.

ARRAY IDENTIFIER =
POINTER TO FIRST ELEMENT

Thus, any reference to an element of an array could be written as a pointer expression.

one-dimensional array

A reference to an element of a one-dimensional array is converted as follows:

ARRAY_ID [SUBSCRIPT]
is equivalent to
*(ARRAY_ID + SUBSCRIPT)

example

```
int A[20];
int *POINTER_A = A;

/* The initial value of POINTER_A is the starting */
/* address of the array element A[0].             */

   *(POINTER_A + 3) = X;

/* The value of X is assigned to A[3].            */
/* The same could have been achieved by:          */

   POINTER_A[3] = X;
```

pointer arithmetic

Pointer arithmetic is defined such that the increment (*POINTER_A + 3* in above example) is scaled by the storage size of the variable pointed to. Regardless of the type of an array, if you increment a pointer by 1, it points to the next element of the array.

There is only one difference between an array identifier and a pointer. A pointer is a variable, whereas an array identifier is a constant. So *POINTER_A = A* and *POINTER_A + 3* are legal operations, but operations on an array identifier (*A=...*) are not allowed.

multi-dimensional arrays

Multi-dimensional arrays are stored row after row.

Assuming an array is declared as

```
int A[d1][d2][d3]
```

and *P* is a pointer to the first member of the array, then the following formula is used to convert an array reference *A[x][y][z]* into a pointer:

$$\boxed{P = P + (x * d2 + y) * d3 + z}$$

The same principle applies to all multi-dimensional arrays.

example

```
int A[2][3][2];
```

The elements of the array are stored as follows:

```
A[0][0][0]  A[0][0][1]  A[0][1][0]  A[0][1][1]
A[0][2][0]  A[0][2][1]  A[1][0][0]  A[1][0][1]
A[1][1][0]  A[1][1][1]  A[1][2][0]  A[1][2][1]
```

Assuming the pointer *P* contains the address of the first element *A[0][0][0]*, then a reference to *A[1][2][0]* is calculated as follows:

$$P = P + (1 * 3 + 2) * 2 + 0$$

$$\longrightarrow P = P + 10$$

Now *P* contains the address of the 11th element of the array.

array_identifier[]

The size of an array may be omitted in **extern** specifications, parameter declarations (see page 6-3) or if a declaration is followed by an initialisation. For multi-dimensional arrays only the size of the first dimension may be missing.

Pointer Arithmetic

address operator &

The unary operator & (address operator) calculates the address of a data element. The address operator may only be applied to variables and array elements, not to constants or expressions.

indirection operator *

The unary operator * (indirection operator) supplies access to the contents of the data element the pointer is referring to. Assuming a structure and a pointer are declared as

```
struct { int DAY, MONTH, YEAR; } DATE, *P;
```

component reference

then *(*P).MONTH* would designate the second component of the structure. As the primary operator has a higher priority than the indirection operator *, the brackets are necessary (see "Associativity and Priority of Operators" on page 7-10).

> Note
> If you want to refer to a component of a structure, the pointer must be declared with the same type as the structure.

structure pointer operator ->

component reference

The structure pointer operator (a minus sign followed by a greater than sign, offers a shorter way of designating a component of a structure. If P is a pointer to a structure S, then P->COMPONENT_ID points to an individual component of the structure.

P->COMPONENT_ID
is equivalent to
(*P).COMPONENT_ID

examples

```
int A,B, *PA;
PA = &A;

/* The address of A is assigned to the pointer      */
/* variable PA. Remember that the type of the       */
/* variable and the base type of the pointer must   */
/* be the same.                                     */

B = *PA;

/* This statement is equivalent to B = A.           */
```

The following operations on pointers are allowed:

assignments to pointer

- Only assignments of addresses and the value 0 to a pointer are meaningful.

addition/subtraction of integers

- Integer values may be added to or subtracted from pointers. If *P* points to an element *A[n]* then *P+i* points to *A[n+i]*, and *P-i* points to *A[n-i]*. See also example on page 5-3.

comparison with zero

- To find out whether a pointer contains an address or not, it can always be compared with 0 (zero). Zero indicates that the pointer is pointing nowhere.

comparison of pointers

- Pointers pointing to elements of the same array can be compared using one of the following operators:

<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equal
!=	not equal

If, for example, *P1* points to an earlier element than *P2* does (e.g. *P1* points to *A[2]* and *P2* to *A[3]*) then the comparison *P1 < P2* is true.

pointer subtraction

- The subtraction of pointers referring to elements of the same array supplies an **int** value, which is the number of elements between both pointers.

example

```
strlen(S)  /* calculate length of string S */
char *S;
{
  char *P = S;

  while (*P != '\0')
    P++;
  return(P-S);
}
```

At the beginning *P* points to the first character of the string. As long as *P* does not refer to a null character (end of string) it is incremented by 1. *P-S* gives the length of the string.

Pointer Arrays

Provided a two-dimensional integer array and an array of integer pointers are defined as

```
int A[10][10];
int *B[10];
```

Then the declaration for *A* reserves storage for 100 elements, whereas for the pointer array *B* storage for 10 pointers is reserved. If every pointer of *B* refers to an array of 10 elements, another 100 storage units will be needed. This storage can be allocated by calling the library function *malloc*, which is described on page 13-68:

storage allocation

example

```
#define char* malloc();

main()
{
  int *B[10], i;

  for (i=0; i<=9; i++);
  B[i] = (int*) malloc(10*sizeof(int));

  /* if you want to get rid off */
  /* the allocated space... */

  for (i=0; i<=9; i++);
    free(B[i]);
}
```

For each pointer of the pointer array *B* *malloc* allocates 40 bytes, in which you can store 10 integers on a ND-500. As *malloc* returns a pointer to **char**, you have to use a cast construct to convert the result into a pointer to **int**.

Now *A* and *B* can be used in a similar way. Both, A[x][y] and B[x][y] refer to a single **int** value.

A pointer array with the same number of elements as a two-dimensional array needs more storage than the equivalent array. In our example *A* takes 400 bytes, whereas *B* takes 440 bytes.

The main reason for using pointer arrays is that the rows of the array may be of different or unknown length. This is often the case for strings.

The example on the next page shows an alternative method of allocating storage for a pointer array. When declaring an array of pointers you can immediately initialise it and thus allocate storage for it. This applies only for character pointers.

example

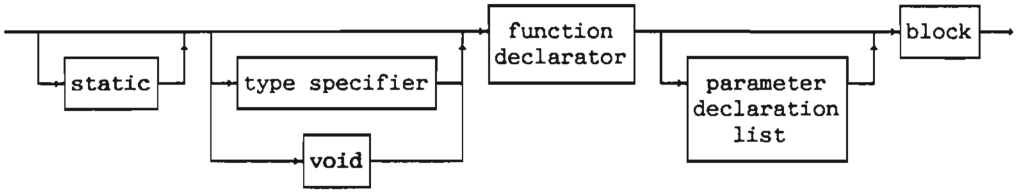
```
/* initialisation of a pointer array */

char *MONTH[] = {
      "January",
      "February",
      "March",
      "April",
      "May",
      "June",
      "July",
      "August",
      "September",
      "October",
      "November",
      "December" };

/* The 12 pointers require 12 * 4 bytes storage,  */
/* whereas the strings require 86 * 1 bytes.      */
/* Thus, this pointer array takes 134 bytes. The  */
/* equivalent two-dimensional array MONTH[12][10] */
/* declared in the example on page 4-11 would     */
/* need 120 bytes.                                */
```

Chapter 6

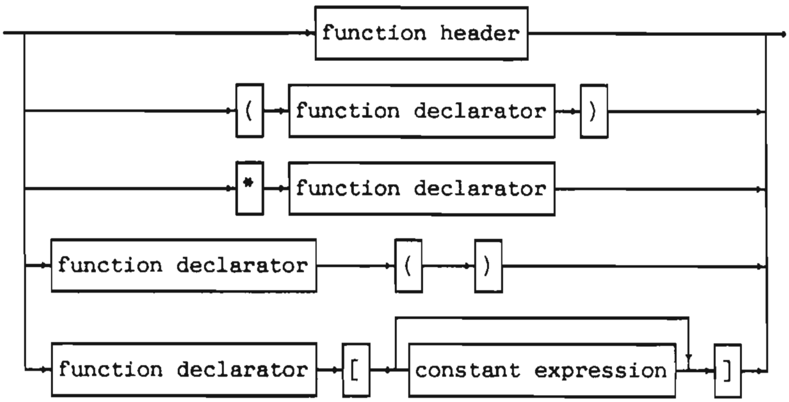
Functions

Syntax of a Function

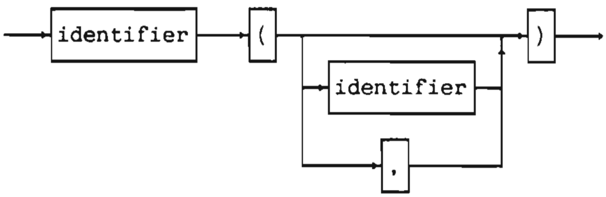
FUNCTION DECLARATION:



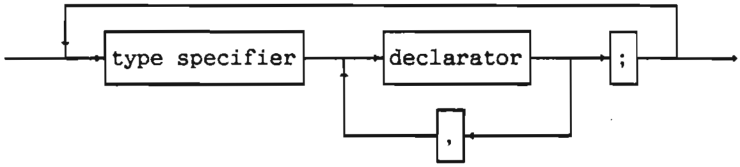
FUNCTION DECLARATOR:



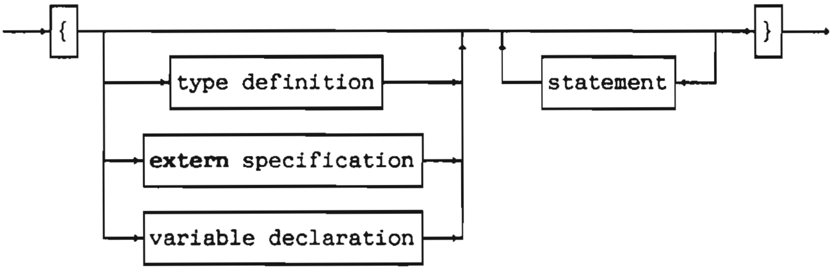
FUNCTION HEADER:



PARAMETER DECLARATION LIST:



BLOCK:



Depending on the result type a function must be declared once or twice:

- If the result type of a function is other than **int**, the function must be declared in the calling function before the call (see page 4-5).

- The second function declaration follows the rules described in the above syntax diagrams. No matter what the result type is, this declaration is always needed.

REMEMBER: Functions have a global scope and cannot be nested. As you can see in the syntax diagrams a function can be declared **static**, thus restricting its scope to its source file.

example

```
main()
{
  double x,f1();       /* type declaration in          */
                       /* calling function             */
  char ch;
  ...
  x = f1(ch);          /* function call                */
}

                       /* function declaration:        */
double f1(w)           /* type and function declarator */
char w;                /* parameter declaration list   */
{
  ...                  /* block                        */
}
```

Parameters

call by value

In C, parameters are passed by value. This means that the values of the parameters passed are copied to the formal parameters,which are local to the function. As an extension to standard C, structures and unions may also be passed as value parameters.

call by reference

However, if you want to change data elements outside the function, you can pass parameters by reference, i.e. you pass a pointer with the starting address of the data element and access the data by indirection. As an array identifier is a pointer expression, array parameters are an example for parameter passing by reference.

type conversions

The following type conversions are implicitly executed before the actual passing of the parameters:

- **float** parameters are converted to **double** values.
- **char** and **short** parameters are converted to **int** values.

It is up to the programmer to ensure that the types of the actual and formal parameters are compatible. If not, you should use a cast construct (see page 3-16).

example

As the function *sin* requires a parameter of type **double**, you should use *sin(1.0)* or *sin((double)1)*, but never *sin(1)*.

variable number of parameters

It is also possible to pass a variable number of parameters. For this purpose the first parameter should indicate how many parameters are passed. It is the programmer's responsibility to take care that the function does not need more parameters than passed and that the data types are compatible.

global variables

If there are too many data which have to be passed, global variables offer an alternative to parameter passing. Depending on their storage class (**static** or **external**) they can be accessed by any function of the same source file or even of the whole program. Another advantage of global variables is that you can initialise arrays and structures.

However, we'd like to repeat that you should use global variables very carefully. Data connections of global variables are not always obvious.

Return Value

Control to the calling function returns either explicitly by the **return** statement or implicitly when reaching the closing right brace of the function.

return

type conversions

The **return** statement followed by an expression in brackets enables you to return a value to the calling function. The value of the expression is the result value of the function. If necessary, the type of the result value is converted to the type of the function (see "Implicit Type Conversions" on page 3-5).

no return value

If the function call is not part of an expression, but a statement, the return value is ignored. If you omit the expression after **return** the return value is undefined.

void

For clarity's sake, a function that does not return a value should always be declared as **void**.

It is not illegal, but bad programming style, if a function, depending on the parameter values, sometimes returns a value and sometimes does not.

Note

The result of a function cannot be an array or a function; pointers to such objects are allowed. As an extension to standard C a function may return a structure or a union (see example on the next page).

examples ●

```
include <stdio.h>       /* for function scanf (see page 13-55) */
main()
{
  int i;
  char S[100];

  /* The integer function pos needs not be declared here. */

  printf ("Input string: ");
  scanf ("%s", S);               /* read input from terminal */

  i = pos(S,"day");
  if (i == -1)
    printf ("The string 'day' does not occur in the word");
    else printf ("The string 'day' starts at position %d\n", i+1);
}


/* pos returns the position of T in S;  */
/* if not found -1 is returned           */

pos(S,T)
char S[],T[];  /* parameter declaration */
{
  int i,j,k;

  for (i = 0; S[i] != '\0'; i++) {
      for (j=i, k=0;  T[k]!='\0' && S[j]==T[k];
           j++, k++)
           ;
      if (T[k] == '\0')
           return(i);
      }
      return(-1);  /* string T not found */
}

/* The first part in the for statement initialises the control */
/* variables, the second part is the condition controlling the */
/* loop. As long as the condition is true, the loop will be    */
/* executed. In the third part the control variables are in-   */
/* cremented by 1. The second for loop does not control any    */
/* statement. If the second return statement was missing       */
/* problems could occur in the main function.                  */
```

```
• struct STR { int a,b;};

  main()
  {
    struct STR S, f();
    S = f(3);
    printf ("a = %d, b = %d\n", S.a, S.b);
  }

  struct STR f(i)
  int i;
  {
    struct STR TEMP;
    TEMP.a = i;  TEMP.b = i*i;
    return TEMP;
  }

  Output: a = 3, b = 9
```

Recursion

In C, functions may also be used recursively, i.e. a function may call itself. When a function calls itself, each invocation gets a fresh set of all automatic variables, independent of previous invocations.

Generally, recursion saves no storage, since all values processed have to be maintained on a stack. But the code of recursive functions is more compact and often easier to understand.

example

```
#include <stdio.h>
main()
{
  int N;
  printf ("Your input number: ");
  scanf ("%d", &N);
  printf ("\nFactorial of %d = %d", N, fac(N));
}

/* fac(N) calculates the factorial of N. (The fac-*/
/* torial is defined only for positive integers.) */

fac(N)
int N;
{
  if (N < 2) return (1);
  else return (N * fac(N-1))
}

/* The factorial for 3 is calculated as follows:   */
/* fac(3) = 3*fac(3-1) = 3*2*fac(1) = 3*2*1        */
```

Chapter 7

Operators and Expressions

Operators

We can distinguish three kinds of operators:

primary operators

- Primary operators can be used in primary expressions. These are references to objects, constants or components of a structure or union, as well as subscripting and function calls. Primary operators are:

  `()    []    .    ->`

unary operators

- Unary operators have only one operand. They are either used as a prefix (before their operand) or as a postfix (after their operand). Unary operators always have a higher priority than binary operators.

binary operators

- Binary operators have a left and a right operand.

If necessary implicit type conversions will be done according to the rules described on page 3-5.

At the end of this section you will find a complete overview of operators' priority.

Arithmetic Operators

OPERATOR	MEANING	TYPES OF OPERANDS
-	unary minus	simple types
+ -	addition subtraction	simple types, pointers (page5-6)
*	multiplication	simple types
/	division	simple types
%	modulus (rest of division)	integer types

The unary minus has the highest priority of arithmetic operators the multiplicative operators *, / and % have a higher priority than the additive operators + and -.

Increment and Decrement Operators

Operators

OPERATOR	MEANING	TYPES OF OPERAND
++	increment, adds 1 to its operand	simple types, pointers (see page 5-6)
--	decrement, subtracts 1 of its operand	

Increment and decrement operators are unary operators. As a prefix operator they increment/decrement their operand before its value is used, as a postfix operator they increment/decrement their operand after its value has been used. These operations are only possible on variables (having an address) and not on constants.

examples

Assuming that *N* equals 5, the statements below assign the following values to *X* and *N*:

statement	value of X	value of N
X = ++N;	6	6
X = --N;	4	4
X = N++;	5	6
X = N--;	5	4

Relational Operators

OPERATOR	MEANING	TYPES OF OPERANDS
==	equality	simple types, pointers, structures
!=	inequality	
>	greater than	simple types, pointers
>=	greater than or equal	simple types, pointers
<	less than	simple types, pointers
<=	less than or equal	simple types, pointers

Comparisons supply the **int** value 0, if the relation is false, and 1, if the relation is true.

Logical Operators

OPERATOR	MEANING	TYPES OF OPERANDS
&&	logical AND	The operands may be of any type, but must be comparable to 0.
||	logical OR	
!	logical negation	

AND &&

The result of a logical AND operation is 1, if both operands are non-zero otherwise the result is 0.

OR ||

The result of a logical OR operation is 1, if either of its operands is non-zero; otherwise the result is 0.

negation !

The unary operator ! returns 0, if its operand is non-zero and 1, if its operand is 0.

Logical expressions are evaluated from left to right, but only until the result is known. The result type is **int**.

In conditions, 0 is interpreted as false and any value different from 0 as true.

Bitwise Logical Operators

OPERATOR	MEANING	TYPES OF OPERANDS
&	bitwise AND	
|	bitwise OR	simple types
^	bitwise exclusive OR	except
<<	left shift	**float and double**
>>	right shift	
~	one's complement	

bitwise AND &

The bitwise AND operator & is used to set bits to zero. The result is the bitwise AND function of the operands.

example

```
C = N & MASK;
```

This statement sets only those bits in *C* to one, that equal one in N <u>and</u> MASK:

```
If    N    = 1101
and   MASK = 1010
then  C    = 1000
```

bitwise OR |

The bitwise OR operator | is used to set bits to one. The result is the bitwise OR function of the operands.

example

```
C = N | MASK;
```

This statement sets only those bits in *C* to zero, that equal zero in N <u>and</u> MASK:

```
If    N    = 1101
and   MASK = 1000
then  C    = 1101
```

bitwise exclusive OR ^

The result of a bitwise exclusive OR operation is one, if the corresponding bits are different, otherwise zero. This operation is also called addition modulo 2.

example

```
C = N ^ MASK;
```

Only if the corresponding bits in N and MASK differ, the bit in C will be set to one:

```
If    N    = 1101
and   MASK = 1010
then  C    = 0111
```

shift operators <<, >>

The expression E1<<E2 shifts the bit pattern of E1 E2 bits to the left. The vacated right bits are filled up by zeros. The expression E1>>E2 shifts the bit pattern of E1 E2 bits to the right. If E1 is unsigned, the vacated left bits are filled up by zeros (logical shift). Otherwise, if E1 is a signed integer value, the shift is arithmetic.

The right operand is converted to an **int** value and the result has the type of the left operand. The result is undefined, if the right operand is negative, or if the length of the left operand in bits is less than the value of the right operand.

one's complement ~

The unary operator ~ supplies the one's complement of an integer. It sets each 1-bit to 0 and vice versa.

Assignment Operators

simple assignment operator

The simple assignment operator = assigns the value of the right operand to the left operand. The left operand must be an expression referring to a manipulatable region of storage. If both operands are arithmetic types the type of the right operand will be converted to the type of the left operand before the assignment. Contrary to other programming languages, assignment operations are also allowed in expressions (e.g. parameter expressions, array subscripts, arithmetic expressions, etc).

> **Note**
>
> One of the most popular errors is to mix up the simple assignment operator = and the comparison operator ==. For example, in the **if** clause
>
> **if** (x=1)
>
> the result is always true, because **x** is assigned the value 1 instead of being compared to 1.

compound assignment operators

The simple assignment operator may be combined to a compound assignment operator (op=) with one of the following binary operators:

```
+   -   *   /   %   <<   >>   &  ^   |
```

```
     E1  op=  E2

  is equivalent to

E1  =  E1  op  (E2)
```

> **Note**
> When using a compound assignment operator E1 is only evaluated once. Furthermore, the brackets around E2 are necessary.

types of operands

The left operand of the += and -= operators may be a pointer. For all other assignment operators the operands must be simple types.

examples

```
   A[i++]  +=  3;

  is equivalent to

A[i++]  =  A[i]  +  3;
```

```
  x *= y + 3;

is equivalent to

 x = x * (y + 3);
```

```
    a[i+j*n] += b[i];

    is equivalent to

a[i+j*n] = a[i+j*n] + b[i];
```

Conditional Operator

In a conditional expression

```
EXPRESSION1 ? EXPRESSION2 : EXPRESSION3;
```

E1 ? E2 : E3

EXPRESSION1 is evaluated first. If it is non-zero the result is the value of the second expression, otherwise that of the third expression. Only one of the second and third expressions is evaluated.

The operands may be simple types, structures or pointers. Remember that a pointer can only be compared to a pointer of the same type or to zero. The usual conversions will be performed to bring the second and third expression to a common type, which will be the type of the result as well. If one of the operands is a pointer the result type is also a pointer.

examples

- The following statement assigns the maximum of *Y* and *Z* to *X*:

  ```
  X = (Y > Z) ? Y : Z;
  ```

  It is equivalent to:

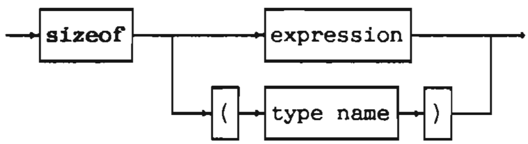
  ```
  if (Y > Z)
     X = Y;
   else
     X = Z;
  ```

- ```
  printf ("%d %s", i, column==80 ? "\n" : " ");
  ```

Sizeof Operator

The unary operator **sizeof** is used to determine the size of an object in bytes. The object may be a variable, an array, a structure or the name of a simple type or structure.

<u>SIZEOF EXPRESSION:</u>

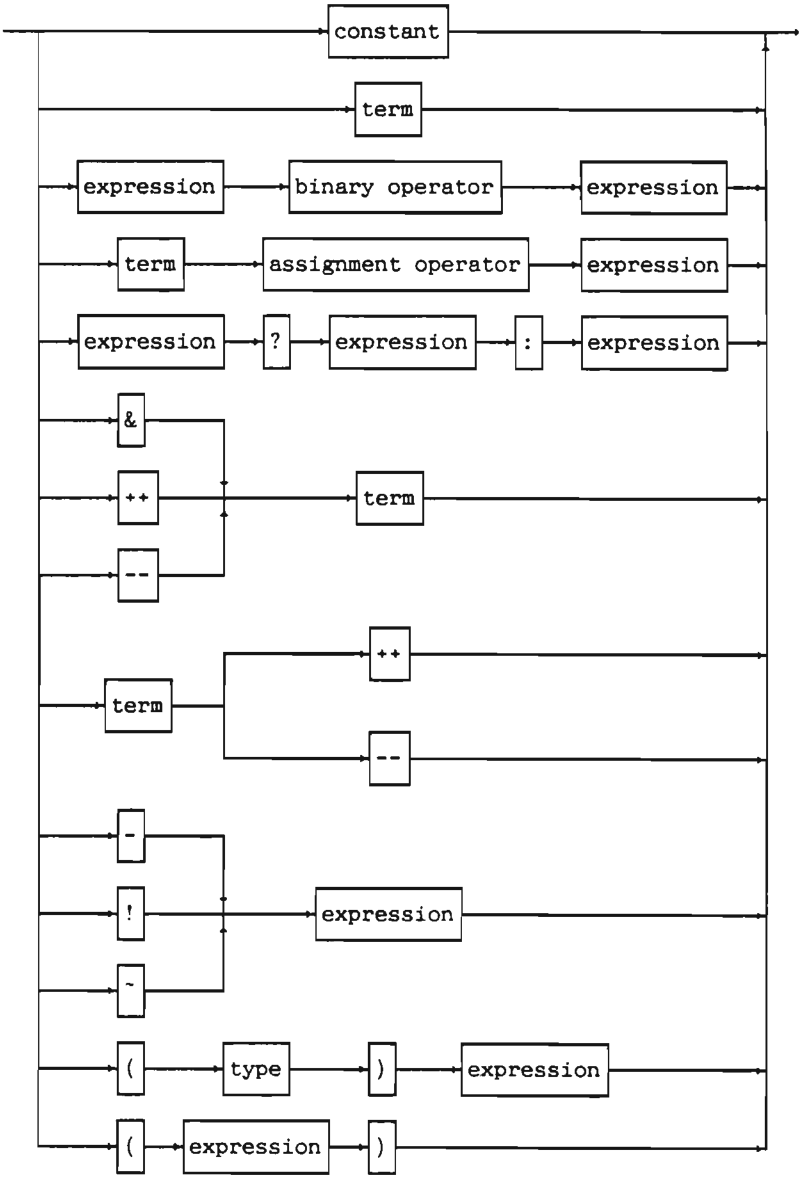
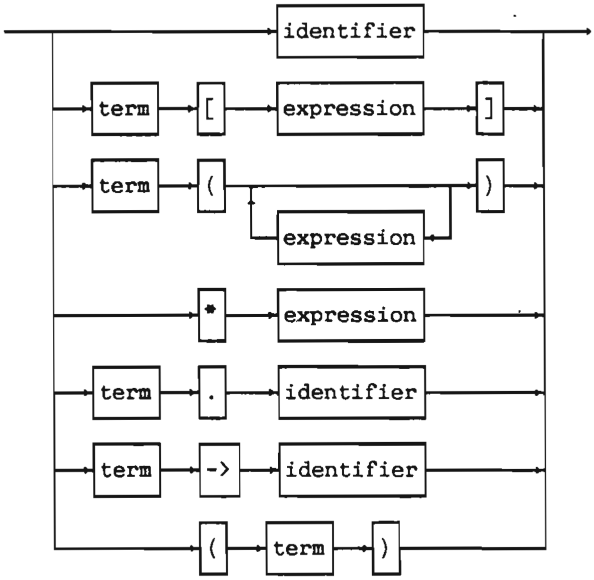
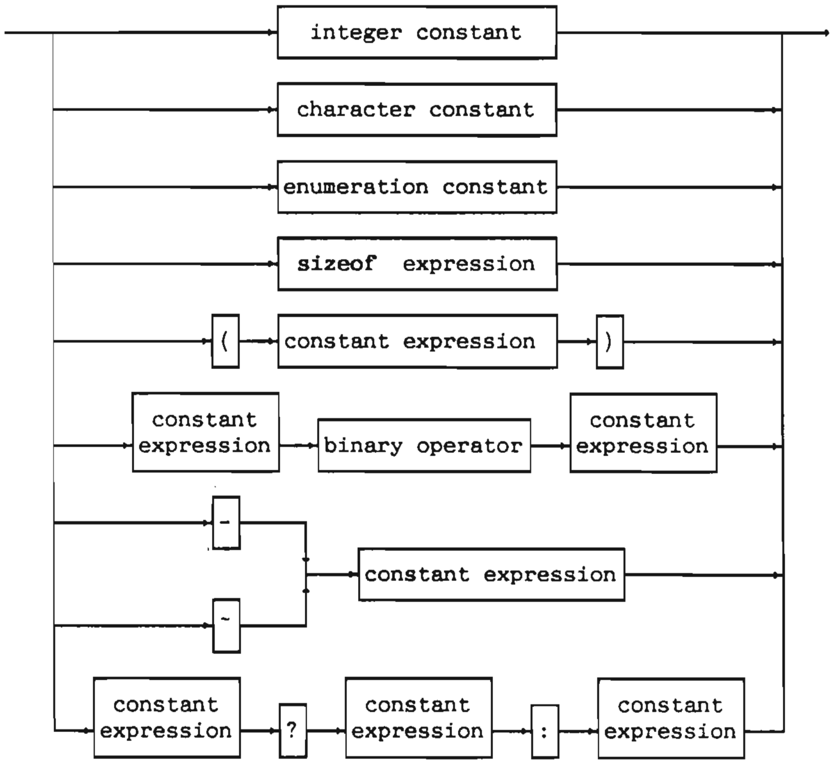
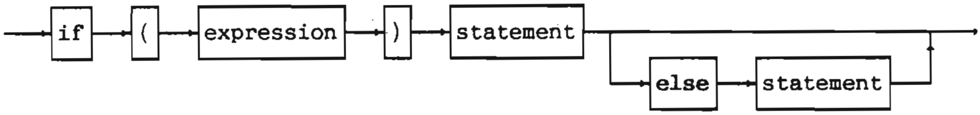


The syntax of an expression is described on page 7-13.
  ```

The result type of a **sizeof** operation is **int**. A **sizeof** expression can be used anywhere an integer constant is allowed. Its major use is in communication with storage allocation and I/O functions.

examples

```
struct S1 {
      char comp1;    /* char takes 1 byte        */
      char *comp2;   /* a pointer takes 4 bytes */
      } arr[] = {
           '1',"A",
           '2',"B",
           '3',"C",
           '4',"D" };

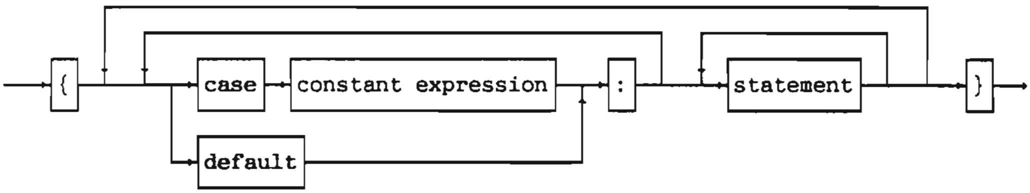
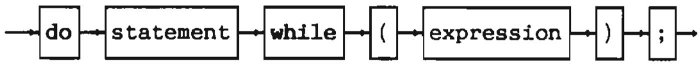
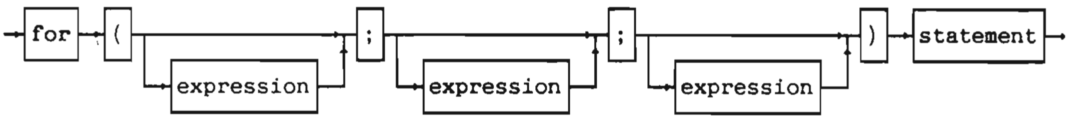
int NRLINES;

NRLINES = sizeof(arr) / sizeof(struct S1)
           /* NRLINES now contains the number of */
           /* reserved rows (20/5 = 4).          */
```

Comma Operator

Two expressions separated by a comma are evaluated from left to right. The type and value of the result are the type and value of the right operand. Most often the comma operator is used in **for** statements.

examples

The following statement is taken from the example on page 6-7:

```
for (j=i, k=0;  t[k]!='\0' && s[j]==t[k]; j++, k++)
```

> **note**
> The comma separating function parameters, variables in declarations, etc. is not a comma operator.

Associativity and Priority of Operators

Operators can group their operands left-to-right or right-to-left. Left associativity (left-to-right) means that brackets are implicitly set from the left and vice versa.

example

Additive operators group left-to-right, therefore:

```
a + b - c + d

is equivalent to

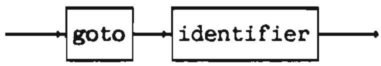
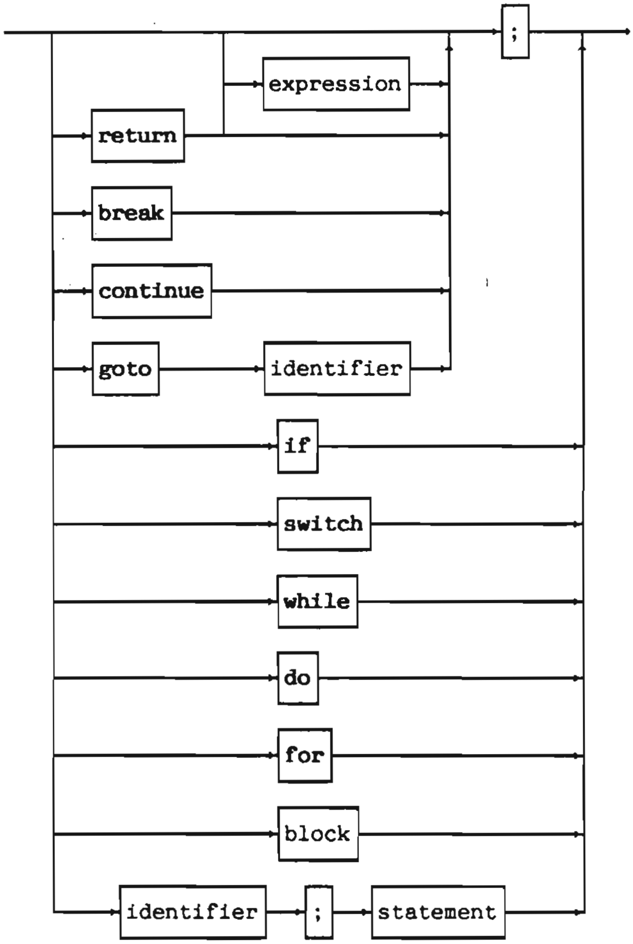
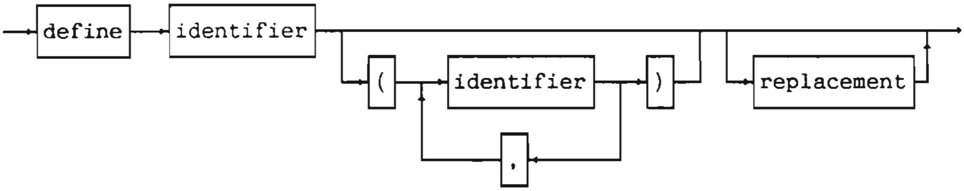
((a + b) - c) + d
```

The table on the next page lists the operators in the sequence of their priority, starting with the highest priority.

The operator (type) in the second line of the table represents the cast construct as described on page 3-16.

ASSOCIATIVITY AND PRIORITY OF OPERATORS

<table>
<tr><th colspan="2">OPERATOR</th><th>ASSOCIATIVITY</th><th>PRIORITY</th></tr>
<tr><td>primary</td><td>() [] -> .</td><td>left-to-right</td><td>highest priority</td></tr>
<tr><td rowspan="2">unary</td><td>(type) sizeof</td><td>right-to-left</td><td></td></tr>
<tr><td>* & - ! ~ ++ --</td><td>right-to-left</td><td></td></tr>
<tr><td rowspan="15">binary</td><td>* / %</td><td>left-to-right</td><td></td></tr>
<tr><td>+ -</td><td>left-to-right</td><td></td></tr>
<tr><td><< >></td><td>left-to-right</td><td></td></tr>
<tr><td>< <= > >=</td><td>left-to-right</td><td></td></tr>
<tr><td>== !=</td><td>left-to-right</td><td></td></tr>
<tr><td>&</td><td>left-to-right</td><td></td></tr>
<tr><td>^</td><td>left-to-right</td><td></td></tr>
<tr><td>|</td><td>left-to-right</td><td></td></tr>
<tr><td>&&</td><td>left-to-right</td><td></td></tr>
<tr><td>||</td><td>left-to-right</td><td></td></tr>
<tr><td>?:</td><td>right-to-left</td><td></td></tr>
<tr><td>= += -= *= /= >>= <<= &= ^= |=</td><td>right-to-left</td><td></td></tr>
<tr><td>,</td><td>left-to-right</td><td>lowest priority</td></tr>
</table>

Expressions

Now that we have explained all operators we can define the syntax of an expression:

EXPRESSION:



A term is defined as:

TERM:

identifier

array

function

pointer

structure reference

structure pointer reference



order of evaluation

The order of evaluation of an expression depends on the priority and associativity of its operators. If operators of the same priority are involved, the order of evaluation is undefined, which means that side effects could occur (e.g. by assignments or function calls). Expressions involving commutative and associative operators (*, +, &, |, ^) may be rearranged by the compiler, even if brackets were used. To force a particular order of evaluation you should use assignments to temporary variables.

type of an expression

The type of an expression is determined according to the rules of implicit type conversion (see page 3-5).

constant expression

A constant expression is an expression that involves only constant integer or **char** values. Such expressions are evaluated at compile time, rather than at run time. They may be used anywhere a constant is required, e.g. as an initialiser.

CONSTANT EXPRESSION:



The constants used in the above syntax diagram are described on page 2-7.

> **Note**
> The binary operator in the middle of the above syntax diagram must not be comma (,).

example

```
arr [100+ (sizeof(int)==4) ? 4 : 2]
```

Chapter 8

Program Structure and Control Flow

Program Structure

In C, the format of a program is free. This means, that several statements may be written on one line; or, one statement may be spread over several lines. The end of a statement is recognised by a semicolon.

statements

functions

As functions may not be declared within other functions, C has no block structure like, for example, PASCAL. Each program consists of one or more functions, always containing a function called *main*. From the function *main* other functions may be called.

function *main*

> **Note**
> The first function of a program must be called **main**.

compound statement

blocks

The braces { and } are used to group declarations and statements together into one compound statement or block. At the beginning of a block variables may be declared and initialised. This applies for functions as well as for other inner blocks.

The syntax of a block is described in the chapter "Functions" on page 6-4.

In the following sections the d fferent statements will be described in detail. The last section of this chapter gives an overview of all statements.

Expression Statement

The expression statement is the statement used most often. It is an expression, as explained in the previous chapter, followed by a semicolon.

assignments or function calls

Usually expression statements are assignments or function calls.

examples

```
p1 = a; p2 = b; p3 = c;
f1 (p1, p2, p3);
```

If Statement

As in other programming languages the **if** statement is used to decide between two or (if nested) more alternatives.

IF STATEMENT:



If the condition is true, i.e. if the value of the expression is non-zero, the statement after **if** will be executed. Otherwise, if the value of the expression is zero and if there is an **else** part, the statement after **else** is executed instead. If the value of the expression equals zero and there is no **else** part, execution continues with the statement after the **if** statement.

shorter code

Since an **if** statement tests the numeric value of an expression,

```
if (expression != 0)
```

can be abbreviated to

```
if (expression)
```

An **else** part always belongs to the inner **if**. To force a different association you have to use braces.

example

```
if (TRUE) {             /* If TRUE is unequal zero      */
    if (X < Y)          /* and X is less than Y,        */
        A = B;          /* A gets the value of B.       */
}
else                    /* If TRUE equals zero,         */
    A = C;              /* A gets the value of C.       */

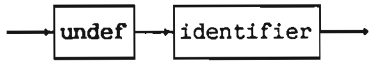
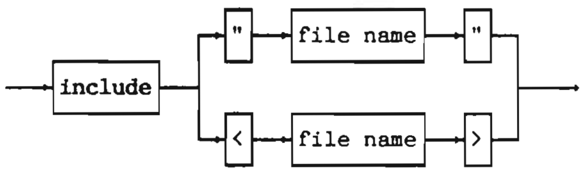
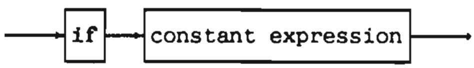
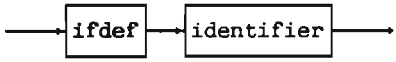
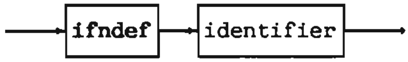
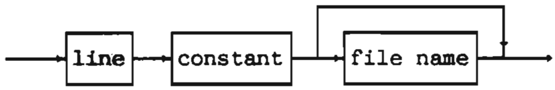
/* The meaning of above sequence changes when           */
/* you leave the braces:                                */

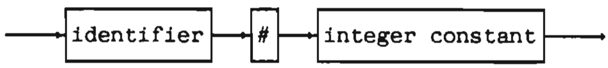
if (TRUE)               /* If TRUE is unequal zero      */
    if (X < Y)          /* and X is less than Y,        */
        A = B;          /* A gets the value of B.       */
    else                /* If TRUE is unequal zero,     */
        A = C;          /* but X is not less than Y,    */
                        /* A gets the value of C.       */
```

Switch Statement

The **switch** statement is another way to select between different alternatives, especially if you want to test whether an integer expression equals one or more constants.

SWITCH STATEMENT:

A **case** statement is defined as:

CASE STATEMENT:



The result of the expressions above must be an integer value (including **char**).

Each of the **case** constants (or constant expressions) may only appear once in a **switch** statement, i.e. the values of the constant expressions must be distinct.

flow of control

The execution of a **switch** statement starts with the evaluation of the expression, which is then compared to all **case** constants. If one of the **case** constants is equal to the value of the expression, control is passed to the statement following this constant. From this statement on, all other statements of the **case** statebe ment will executed and the **case** and **default** prefixes will be ignored. You can leave the **switch** statement by a **break** statement (see page 8-8).

default

If no case matches and there is a **default** prefix, the associated statement will be executed.

If no **case** matches and there is no **default** prefix, the execution of the program continues with the statement after the **switch** statement.

example

```
#include <stdio.h>
main ()
{
  int DIGIT;
  printf ("Please type a number from 0 to 4\n");
  DIGIT = getchar();
  switch (DIGIT) {
    case '0' : printf ("Case 0\n"); break;
    case '1' : printf ("Case 1\n"); break;
    case '2' : ;
    case '3' : printf ("Case 2 or 3\n"); break;
    case '4' : printf ("Case 4\n"); break;
    default  : printf ("Default: not 0..4\n");
    }
  printf ("End of switch statement\n");
}
```

Loops

In C, there are three kinds of loops:

- the **while** statement
- the **do** statement
- the **for** statement

While Statement

The **while** statement is a loop which is executed as long as the value of the expression is non-zero.

WHILE STATEMENT:

The expression is evaluated before the execution of the dependent statement. So, if the condition is false right from the beginning, the statement will not be executed at all.

— Do Statement

Unlike the **while** loop the condition of the **do** loop is tested <u>after</u> the execution of the dependent statement. This means, that the statement is executed at least once.

DO STATEMENT:



— For Statement

The **for** loop is very similar to the **while** loop. However, the **for** loop is often preferred when there is a simple initialisation and reinitialisation. It keeps the loop control statements close together and visible at the top of the loop.

FOR STATEMENT:



initialisation — The first expression is only executed once, and well before the loop starts. Generally, it is the initialisation of the control variable.

condition — The second expression is the condition controlling the loop. It is evaluated before the loop starts. The loop will be repeated as long as its value is non-zero. Its default value is one. This means, when the second expression is missing, you have to leave the loop explicitly with a **break**, **goto** or **return** statement (see pages 8-8, 8-9 and 6-6).

reinitialisation — The third expression is executed after each iteration. In general, it reinitialises the control variable.

As you can see from the syntax diagram all expressions are optional. If you omit one or more expressions do not forget to specify the semicolons.

The statement at the end is mandatory. If there is no dependent statement you must at least specify the semicolon (empty statement).

The following shows the equivalence between a **for** and a **while** loop:

```
for (expression1; expression2; expression3)
     statement

                is equivalent to

            expression1;
            while (expression2) {
                 statement
                 expression3;
                 }
```

example

```
del(S,L)                /* delete L in string S */
char S[];
int L;
{
  int i,j;

  for (i=j=0; S[i] != '\0'; i++)
       if (S[i] != L)
            S[j++] = S[i];
       S[j] = '\0';
}
```

Break Statement

The statement **break;** is used to leave a loop or a **switch** statement immediately. The execution continues with the statement following the innermost surrounding **loop** or **switch** statement. An example is given on page 8-6.

Continue Statement

The statement **continue**; may only be used within a loop (**while**, **do**, **for**). It causes the next iteration of the enclosing loop to begin. For a **while** and a **for** statement this means that the condition is tested again; in a **for** loop the control variable is reinitialised.

Goto Statement

label

In C, each statement may be labelled. A label has the same syntax as any other identifier. It precedes a statement and is followed by a colon.

example

```
PART_1 : for (i=0; i<N; i++) { ... }
```

The **goto** statement causes a jump to a specified label.

GOTO STATEMENT:



Formally, the **goto** statement is not necessary and it is good programming style to avoid it wherever possible.

However, there are a few situations where a **goto** statement may be useful, e.g. to leave two loops at once or to jump to an error handling part.

> **Note**
> A **goto** statement must always be a local jump.
> For global jumps you can use the functions *setjmp* and *longjmp* (see page 13-73).

Syntax of a Statement

STATEMENT:



Chapter 9

The C Preprocessor

Preprocessor Commands

The C preprocessor offers the following extensions to the C language:

- macro definitions (text replacement)
- file inclusion
- conditional compilation
- line control for error handling and debugging
- page skip

starting character #

Lines starting with the character # are recognised as preprocessor command lines. Preprocessor commands are executed before compilation. They have their own syntax, which is independent of the C language; especially they do not end with a semicolon. Furthermore, they may appear anywhere in the program text and are valid from the place of appearance until the end of the appropriate source file (independent of other scope rules).

> Note
> Preprocessor commands in the source file must start with a hash (#).

To see what the C preprocessor does exactly, you can give the compiler command **preprocess**. Your source file will then be output including replacements, include files, etc. More about this in chapter "Compiler commands" starting on page 10-3.

Macros

text replacement

With the command **define** you can define a macro which causes the preprocessor to replace a specified identifier (with an optional parameter list) by a given text. The identifier will be replaced anywhere it appears in the source file, except in strings and comments.

> Note
> Redefinitions of macro identifiers are allowed. If the definitions are not identical, you will get a warning and the last definition will be taken.



The identifiers obey the syntax rules for C identifiers (see page 2-4). The first blank or equal sign following the first identifier is interpreted as a separator between the text to be replaced and the replacement. So, to be recognised as an entity a parameter list must follow the identifier immediately. The number of formal and actual parameters must be the same.

The replacement text is arbitrary. If no replacement is given, the identifier will be replaced by nothing. Such definitions are useful for conditional compilation, when testing whether an identifier is defined or not (see page 9-8).

symbolic constants

Most often **define** commands are used to define symbolic constants at the beginning of the program.

example

```
#define MAXLINES 60

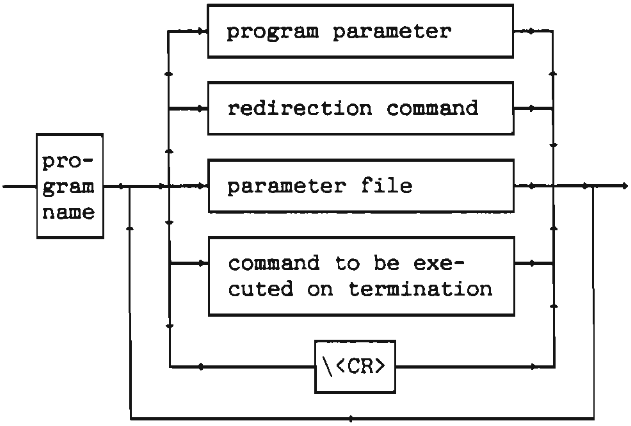
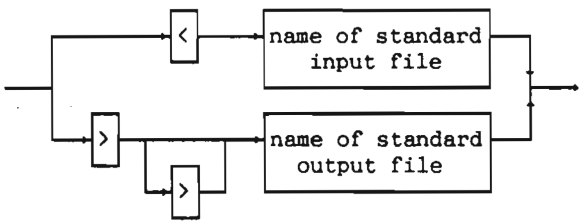
  main()
  {
    int line;
    ...
    if (line > MAXLINES)
    {
      printf("MAXLINES = %d\n", MAXLINES);
      /* As it is part of a string the first       */
      /* MAXLINES in above statement will not be   */
      /* replaced.                                 */
      ...
    }
  }
```

ID(parameter_list) — Below you find an example where the text to be replaced consists of an identifier with a parameter list.

example

```
#define MAX(A,B) ((A) > (B) ? (A) : (B))
```

The line

```
m = MAX(r+s,t+u);
```

will be replaced by

```
m = ((r+s) > (t+u) ? (r+s) : (t+u));
```

side effects — The definition above provides a 'function' that may be used for any data type. To avoid side effects and to ensure the intended order of evaluation you should set brackets very carefully.

continue command on next editor line — If an editor line is too short for a macro definition you can continue on the next line by placing a backslash (\) at the end of the line to be continued.

macros in the user interface — In addition to defining macros in a source file you can also define them directly in your user interface (see also page 11-9). Identifiers of so defined macros will be replaced in any program to be compiled under your user. To enter definitions into this user interface you

1. call the C compiler with the SINTRAN command

   @NC ↵

2. give the **define** command as described above, but leaving the initial #, e.g.

   NC: **define** YES 1 ↵

3. save the macro permanently in your user interface by entering the command

   NC: **save-compile-parameters** ↵

4. activate the definitions by the command

   NC: **initialize-compile-parameters** ↵

If you do not specify a filename after the two commands above the default file NC-A:INIT will be used.

delete a macro

The preprocessor command **undef** deletes a previously defined macro:



From this command line onwards the specified identifier will no longer be replaced. Parameters, if any, need not be specified.

example

```
#define NR 100
main()
int X,Y;
{
  X = NR;    /* NR will be replaced by 100 */
  ...
#undef NR
  ...
  Y = NR;    /* here, NR is not defined as a macro */
  ...
}
```

File Inclusion

An **include** command line will be replaced by the contents of the specified file.



The *file name* may be written in SINTRAN or UNIX notation (see page 13-5).

user / directory

> Note
> A user under SINTRAN, with or without directory specification, corresponds to a directory under UNIX. When talking of users in this manual, we always refer to the SINTRAN user.

file search

If the *file name* in an **include** command does not specify a user, a search according to the following rules takes place:

- If the *file name* is enclosed in quotes ("..") , the file is searched for under the following users in the sequence as stated:

  1. the user under which the main file is compiled
  2. the user under which you are logged in
  3. the user(s) you specified with the compiler command **directory** (see page 11-9)
  4. the C system user CAT-NC-500:
     Under this user so-called *header files* are stored, which contain macros and definitions used by the C library functions (see page 13-3).

C system user CAT-NC-500

- If the *file name* is enclosed in angle brackets, the file is only searched for under:

  1. the user(s) you specified with the compiler command **directory** (see page 11-9)
  2. the C system user CAT-NC-500

example

Each source file containing function calls to the standard I/O library should start with:

```
#include <stdio.h>
```

The file *stdio.h* is a header file stored under the C system user CAT-NC-500 and contains definitions of macros and variables used in the standard I/O library.

Another typical example of an include file is to combine general macros and variable declarations, which can then be used by several source files.

example

Assuming a file *GEN-DEF* is stored under a special user COMMON, which is entered into your user search list with the compiler command **directory**. An **include** command in a program stored under your user would then be coded as:

```
#include <GEN-DEF>
```

An include file may include other files up to a maximum depth of nesting of 10. For clarity's sake, files should always be included at the top of a source file.

> **Note**
> Identifiers of macros defined before an **include** command will be replaced in the included file as well.

Conditional Compilation

With the following set of preprocessor commands you can exclude certain parts of your source file from compilation. This may be useful in order to generate different program versions from the same source.

There are three commands, which test a condition:

1. The command



checks whether the *constant expression* has a non-zero value.

2. The command



checks whether the *identifier* is currently defined in the preprocessor (by a previous **define** command).

3. The command



checks whether the *identifier* is currently undefined in the preprocessor.

After an arbitrary number of lines, each of these three tests may be followed by the command

The whole construct is terminated by the command

If the condition, checked in one of the three tests, is true, i.e.

1. the constant has a non-zero value, or
2. the identifier is defined in the preprocessor, or
3. the identifier is undefined in the preprocessor,

then all lines between an **else** and the **endif** will not be compiled. If the condition is false, all lines between the test and an optional **else** or **endif** will be ignored.

These tests may also be nested.

Line Control

The **line** command is implemented in order to be compatible with other C compilers. Usually, error reports from the compiler refer to the editor line of the source file. The preprocessor command **line** gives a source line an absolute value:



The *constant* in this command defines the number of the next line and will be used as a new base for counting. The optional *file name* may designate the name of the source file. However, under SINTRAN the *file name* is ignored.

Page Skip

The command

causes a page skip in a printout of the source file.

Predefined Macros

In order to facilitate user error handling the following identifiers are predefined in the preprocessor:

- __LINE__
  This identifier will be replaced by the current line number of the source file.

- __FILE__
  This identifier will be replaced by a string containing the current source file name enclosed in quotes.

- __DATE__
  This identifier will be replaced by a string containing the current date enclosed in quotes.

- __TIME__
  This identifier will be replaced by a string containing the current time enclosed in quotes.

- SIN3 and ND500
  These two macros are flags that are set to 1. They can be useful when having a source file which is to be compiled on different machines or under different operating systems. For example, machine-dependent code can be introduced by the preprocessor command

  ```
  # if ND500
  ```

  so that the dependent statements will only be compiled on a ND-500 machine. The equivalent applies to the flag SIN3, which refers to the operating system SINTRAN III.

example

```
main()
 {
 int lineno;
 char *file_name;

  lineno = __LINE__;
  file_name = __FILE__;
  printf ("%s   %s\n",__DATE__, __TIME__);
  printf ("file name = %s\nline number = %d\n", file_name, lineno);
  }
```

The output will look like:

```
Sep-30-86   09:43:55
file name = (DIR-NAME:USER-NAME)OBJECT-NAME:FILE-TYPE;VERSION
line number = 6
```

Chapter 10

Extensions for System Programming

Monitor Calls and Machine Instructions

Machine Instructions

If you want to call monitor calls (SINTRAN system functions) or machine instructions from your C program you have to specify them as external functions (see "**extern** specification" on page 4-8). To distinguish monitor calls and machine instructions from other ordinary functions, the identifier in the **extern** specification must be followed by a hash (#) and an integer constant.

```
──▶[identifier]──▶[#]──▶[integer constant]──▶
```

The hash and the integer constant must only be given in the **extern** specification; in the call itself you just specify the identifier.

The identifier may be freely chosen by you, while the integer constant must represent the number of a monitor call or machine instruction. Numbers less than 1000 refer to monitor calls; all other numbers are taken as numbers of machine instructions.

example

```
main()
{
extern leave-program#0();
extern double n500-sqrt#1204();
...
if error
   leave-program();
...
}
```

> **Note**
> Parameters to monitor calls have to be passed by reference, i.e. they must be pointers or arrays.

Detailed descriptions of existing monitor calls can be found in the SINTRAN III Reference Manual or the SINTRAN III Monitor Calls manual. Machine instructions are described in the ND-500 Reference Manual.

monitor call interface

As the parameter passing in monitor calls is machine-dependent you should try to avoid the way of calling described above. If the monitor call wanted is integrated in the interface for monitor calls (see page 15-3) you should use this one. The monitor call interface guarantees a uniform interface for ND-100 and ND-500, which may be of interest in future releases when C is also available on the ND-100.

Register Variables

Register variables are variables with absolute addresses, as such allowing access to machine registers (see table on the next page). They are declared as any other variable, the only difference being that the variable identifier has to be followed by a postfix, i.e. a hash (#) and an integer constant representing the number of a register. This postfix must only be specified in the declaration of the variable.



When using registers you should take into account that the compiler itself uses the same registers (e.g. for calculating operations or function calls). If, for example, for an assignment a conversion is required, the computer could destroy the registers used. The use of register variables strongly depends on the structure of the machine instructions of the processor. Detailed descriptions can be found in the ND-500 Reference Manual.

ND-500 HARDWARE REGISTERS

NUMBER	REGISTER
0	program_counter
1	L-register
2	B1-register
3	B2-register
4	TOS-register
5	"low-limit-trap"-register
6	"high-limit-trap"-register
7	"trap-hard-address"-register
8-11	4 working registers for integer
12-18	4 working registers for floating point
20	status-register-1
21	status-register-2
22	own-trap-enable-register-1
23	own-trap-enable-register-2
24	mother-trap-enable-register-1
25	mother-trap-enable-register-2
26	child-trap-enable-register-1
27	child-trap-enable-register-2
28	trap-enable-modification-mask-1-reg.
29	trap-enable-modification-mask-2-reg.
30	current-executing-domain-register
31	current-alternative-domain-register
32	process-segment-register

example

In order to examine the first status register you can use the following code:

```
main()
{
int status, machine_status#20;
...
status = machine_status;
if (status == error_code)
   ...
}
```

Stack initialisation

Initstack installs an own stack for a C module. This is only needed when mixing C modules into programs written in another language as for instance COBOL, PLANC or FORTRAN. It will solve problems of stack conflicts or stack overflow.

example

```
static int stack[1000]; /* the size of the stack will be 4000 */
                   /* bytes the stack must be a static   */
                   /* variable                           */
extern void initstack#13100();

void p1()
{
 initstack(&stack,sizeof(stack)); /* parameters to initstack are
                                    1. the address of the stack
                                    2. the size of the stack */
 printf("this is p1\n");
}

main()
{
 printf("main calling p1 with init-stack\n");
 p1();
 printf("back in main \n");
}
```

Chapter 11

Compiling and Linking

Conflicts between C source file names and routine names

To say it short: you should avoid to have a routine name in your C source program which is identical to the file name except for "-" and "_" characters.

argument list

When a C program is started the program name has to be made the first character of the argument list. As under SINTRAN the name of the started program is lost, the program name has to be fixed at compile time. At this time only the name of the source file is known.

The following changes take place when the source file name is converted to the program name. The file extension is left out and all '-' characters are changed to '_' characters with respect to the debugger.

If, after these changes, the name of the program equals that of the a routine, the LINKAGE-LOADER will output a "redefinition ignored" which means that the definition of the routine is lost. When the program is started in spite of the warning the runtime error "instruction sequence order" will occur when the routine with the conflicting name is called.

Compiler Invocation

You invoke the compiler from SINTRAN by giving the command:

```
@NC ↵
```

The compiler prompts with the notification of the version in use and on the next line *NC:*

```
Norsk Data C - Version: A06 - 1989-01-10
NC:
```

command mode

Now you are in compiler command mode, i.e. you can give commands to the compiler. Parameters may be given either in the command line or in the dialogue. Compiler commands and their parameters may be abbreviated to their shortest unambiguous form.

exit

To leave the compiler command mode and return to the operating system you give the command:

```
NC: exit ↵
```

help

The command **help** gives you a list of all available compiler commands, which will be described in the following sections:

```
NC: help ↵
command: ↵
cc
help <command: >
exit
preprocess <source file: >,[<list file: >],[<output file: >]
check <source file: >,[<list file: >],[<CAT file: >]
generate-code <CAT file: >,<object file: >
compile <source file: >,<list file: >,<object file: >
link <source file: >,<program: >
cross <source file: >,<cross reference file: >,<lines per page: >
format <source file: >,<new source file: >
value <definitions / options / libraries: >
define [<macro identifier [(identifier,...)]: >],[<replacement: >]
undef [<macro identifier: >]
directory [<include directory/user: >]
options <option: >...
page-length [<lines: >]
library <library file: >...
initialise-compile-parameters [<initialisation file>]
save-compile-parameters [<initialisation file: >]
clear
@<SINTRAN-command>
```

Compiling a Program

There are several possibilities:

- You can have your program processed by the preprocessor only.
- You can compile your program without producing an object file. Instead of this a temporary CAT file containing intermediate code will be produced, which may be used afterwards to generate the object code.
- You can compile your program and produce the object code with a single command.
- You can compile and link your program with a single command.

Preprocess

When giving the command

preprocess <source file: >,[<list file: >],
[<output file: >]

the source file will only be processed by the C preprocessor, i.e.

- The syntax of preprocessor commands will be checked.
- Macro identifiers will be replaced wherever they occur.
- **include** commands will be replaced by the contents of the file specified.

source file — Here you have to state the name of the source program to be processed. The default types of the **source file** are :C and :SYMB, :C being the primary type.

list file — The **list file** will contain error messages of the preprocessor. If no file is specified the terminal will be taken as output device.

output file — If no output file is specified, the output will be written to the terminal. The output file does not contain source file comments.

Check Source Code

The command

check <source file: >,[<list file: >],[<CAT file: >]

compiles the program specified as source file, but does not produce an object file.

source file — The **source file** is the name of the program to be compiled. The default file types of the *source file* are :C and :SYMB, :C being the primary type.

list file — The *list file* will contain error messages of the compiler. If no file is specified the terminal will be taken as output device. If a *list file* is specified and the program is compiled with option *a+* (see page 11-12) it will also contain a program listing.

CAT file — The *CAT file* is a temporary file containing intermediate code. It may be used as input file for the command **generate-code** (see below). If no *CAT file* is specified, the output will be written to a temporary file named *SCRATCH-OONNN:CAT*, where *NNN* is your terminal number. This file will be overwritten by the next **check** or **compile** command.

Generate Code

The command

generate-code <CAT file: >,<object file: >

takes the specified *CAT file* which results from a previous **check** command and produces an object file. If no *CAT file* is specified the file *SCRATCH-OONNN:CAT* will be taken as input. If no object file name is given, the object code will be written to a temporary file named *SCRATCH-OONNN:NRF*, where *NNN* is your terminal number.

This command is particularly implemented for future releases when operations on the intermediate code will be possible.

Compile

The command

compile <source file> <list file> <object file>

compiles the program specified as *source file* directly producing the object code.

source file — The *source file* is the name of the program to be compiled. The default file types of the *source file* are :C and :SYMB, :C being the primary type.

list file — The *list file* will contain error messages of the compiler. If no file is specified the terminal will be taken as output device. If a *list file* is specified and the program is compiled with option *a+* (see page 11-12) it will also contain a program listing.

object file — The *object file* is the file in which the object code will be stored. The default type of the *object file* is :NRF. If no object file name is given, the output will be written to a temporary file named *SCRATCH-OONNN:NRF*, where *NNN* is your terminal number.

Compile and Link

The command

link <source file: >,<program: >

compiles and links the program specified under *source file*. Libraries which have to be loaded in addition to the C library can be specified in the user interface with the command **library** (see page 11-13).

source file — The *source file* is the name of the program to be compiled. The default file types of the *source file* are :C and :SYMB, :C being the primary type.

program — Here you specify the name of the executable program. It will be stored in a domain with the name specified.

Source File Listing

The command

cross <source file: >,<cross reference file: >,
<lines per page: >

produces a formatted program listing of the specified *source file*. Each page of the listing starts with two header lines containing the actual date and time and information about the file. If the number of *lines per page* is not given, the value specified with the compiler command **page-length** (see page 11-12) or the default value of 48 lines will be taken.

In the left margin of the listing the level of nesting, the source line number and the program line number are given. The source lines will be indented according to the level of nesting.

structuring errors

Errors in the block structure will be indicated with the message ERROR IN BLOCKSTRUCTURE at the place of occurrence. If no error is found, the message NO ERROR IN BLOCKSTRUCTURE will be output at the end of the file.

If the *cross reference file* is a printer, keywords will be bold printed.

format source file

The command

format <source file: >,<new source file: >

produces a formatted source file, where the lines are indented according to the level of nesting. Structuring errors will not be reported.

The *new source file* must be different from the old *source file*.

Compile Parameters

Definitions

The command

value definitions

gives a list of all macro definitions and directory specifications currently defined by you in the user interface.

define, undef

As you already know, macros cannot be defined and deleted in a source file only, but also in your user interface (see description of the commands **define** and **undef** beginning on page 9-3). Given as a compiler command in your user interface these commands must not start with a hash (#).

directory

With the command

directory [<include directory/user: >]

you can specify a SINTRAN user (or UNIX directory) for default file searching. The sequence of searching is described on page 9-7.

Options

The command

value options

gives a list of all options in the user interface with their current setting. The list below shows their default values:

default options

```
NC: value options ↵
options:
  target machine is ND-500       (m2)
  4 byte record alignment        (a4)
  double arith. for floats       (f-)
  64 bit real                    (r4)
  with line numbers              (l+)
  with symbolic debug            (d+)
  with procedure names           (n+)
  without subrange check         (s-)
  without pointer check          (p-)
  without index check            (i-)
  without overflow check         (o-)
  without profiling              (pr-)
  externals as common            (ic+)
  with library mode              (lm+)
  without trace                  (t-)
  without complete listing       (a-)
  with local optimization        (lo+)
  page length is 48 lines
```

change option value

You can change the value of an option by the command

options <option: >...

To switch on an option you specify the letter of the option wanted followed by a plus sign.

To switch off an option you specify the letter of the option wanted followed by a minus sign.

EXCEPTIONS: Do not change options *m* and **r**. They are implemented for future releases only.

If you want to change the values of several options with one command, you have to separate them by a comma or a blank.

example

With the following command you activate option *pointer check* and reset option *line numbers*:

NC: **options** p+,l-↵

record alignment (ax)

Option ax provides the opportunity to align records on one-byte (x=1), two-byte (x=2) or by default four-byte (x=4) boundaries and thus to optimise the code generated for the ND-5000 computers. In the A00-version of the NC compiler default was one-byte alignment.

float arithmetic (f)

C defines that all float operations are carried out with double precision, i.e. before computation all float variables are converted to double whereas the result is converted to float. With the option *f* set to *f+* these conversions are omitted except for actual function parameters thus speeding up float operations. With the option set to f- the float-to-double conversion is performed.

line numbers (l) — If a program is compiled with option *l+*, the number of the error line will be output when a runtime error occurs. The line numbers are stored in a table generated by the compiler, which is part of the data area; the program area and thus the execution time remains unchanged.

symbolic debug (d) — When executing a program compiled with option *d+*, the runtime system generates symbolic debug information (see Symbolic Debugger User Guide).

procedure names (n) — If a program is compiled with option *n+*, the name of the function in error will be output when a runtime error occurs. Like line numbers, name information is stored in the data area of the program.

For a better error check the following four options (*s*, *p*, *i* and *o*) should be switched on when compiling a program for the first time. As they generate additional code, they should be switched off before the final compilation of the program. This makes the executable program smaller and faster.

subrange check (s) — When executing a program compiled with option *s+* the runtime system will check whether values of variables on the left side of an assignment exceed their ranges. For example, if you assign an integer value greater than +32767 to a short variable, the program will abort with the message "subrange or index out of range".

pointer check (p) — When executing a program compiled with option *p+*, the runtime system checks, whether pointers used as references are unequal 0 and point to a legal address. If not, the program will abort with the message "pointer with nil value".

index check (i) — When executing a program compiled with option *i+*, the compiler checks whether array indices are in the defined range. If not, the program will abort with the message "subrange or index out of range". Pointer arithmetic to access arrays is not checked by this option.

overflow check (o) — When executing a program compiled with option *o+*, it is checked whether intermediate arithmetic results exceed the range of 4-byte integers or double values. When an overflow occurs the program aborts with the message "real arithmetic overflow".

profiling info (pr)

If option *pr* is set to *pr+*, at runtime information about the procedures' and functions' calling hierarchy is written to a file. This information is evaluated by a program called CAT-Profile (ND-no. 211565). The profile information is only of use with this new product.

external variables (ic)

If you want to export variables to or import variables from FORTRAN *ic* should be set to *ic+*. In this case exported and imported variables are treated as FORTRAN common blocks like they were in all NC versions before A06.

With the option set to ic- exported and imported variables are handled in the way PLANC does. If you want to export variables to or import variables from PLANC the option should be set to ic-.

If option *ic-* is set variables will be handled as described in the following table

declaration	handled as
extern int i;	import
int i = 0;	export
int i;	common

If a variable is not initialized in the declaration part it is treated as a common variable. It has to be declared in each module where it is used.

If it is declared with the storage class identifier extern it will be treated as an imported variable. Each imported variable must be exported from and initialised in one and only one module.

If it is declared without a storage class identifier and initialised in the declaration part it is treated as exported variable. In this case it must be imported by another module.

library mode

With option *lm+* (default) library marks are written to all modules. Option lm- (no library marks) allows the linkage-loader facility *reload* to be used.

trace (t)

When being compiled with option *t+* the progress of compilation is reported to the user. The report appears on the specified *list file*, which is the terminal by default.

complete listing (a)

With option *a+* the compiler writes a source listing to the specified *list file*. This source listing includes error messages at the places where errors occur.

local optimisation (lo) Option lm+ (default) causes an inlinecall instruction for the library routine strcpy to be used which is much faster than any software routine or macro. If your program contains conditional expressions like "(expr ? strcpy(..) : "string")" option *lo* has to be set to *lo-* as the inlinecall wouldn't work in such a case.

page length With the command

page-length [<lines: >]

you can change the page length for printer output. The default length is 48 lines.

Libraries

With the command

value libraries

you can list the libraries defined in your user interface, and which are loaded in addition to the C library when giving the compiler command **link**. You can define additional libraries in the user interface with the command

library <library file: >...

The *library file* must be a file of type :NRF.

Initialise the User Interface

You can save the current compile parameters in an initialisation file with the command

save-compile-parameters [<initialisation file: >]

The *initialisation file* will contain the current setting of the compile parameters (definitions, options, libraries). Its default file type is :INIT. If no name is specified, the values will be stored in a file named *NC-A:INIT*.

You can create different initialisation files, each for a different purpose. To activate a certain set of compile parameters you give the command

initialise-compile-parameters [<initialisation file:>]

In subsequent compilations, the parameter values stored in the *initialisation file* specified will be used.

When invoking the compiler an initialisation file will be loaded automatically:

1. First the compiler tries to load the file *NC-A:INIT* of your own user.

2. If the file *(OWN-USER)NC-A:INIT* is not present, the compiler tries to load the initialisation file of user SYSTEM *(SYSTEM)NC-A:INIT*.

3. Otherwise, if the system initialisation file is not present either, the compiler defaults as described on page 11-9 will be taken.

The compiler command

clear

resets the compile parameters to its defaults. All macro definitions, directories and libraries currently active in your user interface will be deleted and the option values will be reset to the compiler defaults described on page 11-9.

Comments

When using a mode job for the compilation or linking of your program, the command **cc** introduces a comment line. (See also SINTRAN III Time Sharing/Batch Guide.)

example

Enter your mode job COMPILATION in PED:

```
@cc    **************************
@cc    *      compilation       *
@cc    **************************
@NC
initialise-compile-parameters init-file
cc compile my-source into my-object
compile my-source,,my-object
exit
```

Activate your mode job from SINTRAN:

@mode COMPILATION,,↵

SINTRAN commands

When starting a compiler command line with the SINTRAN prompt sign you can give commands to the operating system.

example

```
@list-files,:c,,
```

Linking a Program

Instead of linking your program with the compiler command **link** you can also link it with the ND-500 linkage loader. In order to link a C program you have to load at least two libraries:

```
    the C library              :  NC-LIB:NRF
and the multi-language library:  CAT-LIB:NRF
```

On the ND-500, a program is not loaded into a program file, but into a domain. If you do not define (or set) a domain, the executable program is loaded into the temporary domain SCRATCH-DOMAIN and will be overwritten by the next loading process without a domain name.

order of loading

When linking a C main program the following sequence must be observed for the loading of functions and libraries:

1. C main program
2. optionally: external routines/functions (C, FORTRAN and/or PLANC)
3. C library NC-LIB:NRF
4. CAT library CAT-LIB:NRF
5. optionally: ISAM library ISAM-LIB:NRF
SIBAS library SIBAS-LIB:NR F
FOCUS library FOCUS-LIB:NR F
6. optionally: FORTRAN library FORTRAN-LIB: NRF
PLANC library PLANC-LIB:NR F

If the optional libraries exist only as sharable segments (depending on the installation) the corresponding segments have to be linked.

Detailed descriptions of the linker and linking process can be found in the ND-500 Loader/Monitor manual.

example:

Terminal input/output	Remarks
@ND-500-MONITOR LINKAGE-LOADER	Call linkage loader
ND-Linkage-Loader-X	
NLL: DELETE-AUTO-LINK-SEGMENT NLL: DELETE-AUTO-LOAD-FILE	To avoid loading of FORTRAN library
NLL: SET-DOMAIN "test"	Name of domain (SINTRAN notation)
NLL: OPEN-SEGMENT "test"	If you want to recompile and link your program via the *link* command.
NLL: LOAD-SEGMENT test	Load main program
	test:NRF
Program: ....xxxxxxP01 Data: xxxxxxD01	Free storage
NLL: LOCAL-TRAP-DISABLE all	To avoid undefined references for trap handling
NLL: TOTAL-SEGMENT-LOAD test-module	Load module(s)
Program: ....xxxxxxP01 Data: xxxxxxD01	Free storage
NLL: LOAD-SEGMENT NC-LIB:NRF	Load C library
Program: ....xxxxxxP02 Data: xxxxxxD02	Free storage
NLL: LOAD-SEGMENT CAT-LIB:NRF	Load CAT library
Program: ....xxxxxxP02 Data: xxxxxxD02	Free storage
NLL: DEFINE-ENTRY stack-space,400000,d	Define stack size: 400000B default = 128K bytes
NLL: DATA-REFERENCE stack-space,rts stack size,d	
NLL: LIST-MAP Unsatisfied references:	List references

```
None!

Defined symbols:                                    List of defined entries
  .
  .
Program: ....xxxxxx P  Data: ....xxxxxx D

NLL: EXIT                                           Return to SINTRAN
```

Chapter 12

The Command Line

General

Before a C program starts execution of the function *main*, the runtime system interprets the whole command line.

Everything to be considered concerning the SINTRAN command line will be described in the following sections.

Command line syntax



Command line interpretation

The command line syntax is different between SINTRAN and the C runtime system. As there is nothing like a UNIX shell in SINTRAN C has to emulate a shell in order to be able to interpret a C command line.

Continuation lines

Reading of input is continued on the next line, if the runtime system finds a "\" (backslash) followed by a carriage return. On the continuation line you are prompted for input by a ">" (greater than) character.

example

@<u>ND-500 myprog\↵</u>
><u>here are six more input parameters↵</u>

Execute command after termination

A command can be specified in the parameter list which will be executed after program termination. A ";" (semicolon) as first character identifies the following characters as a command to the runtime system. The command has to be enclosed by quotation marks, if you want to pass parameters to the command. A semicolon can not be passed as first character of a parameter.

examples

```
@cc call myprog with arg1 and arg2 and start PED after
@cc program termination
@nd-500 myprog arg1 arg2 ;ped
@cc same as above but call PED with file file1:symb
@nd-500 myprog arg1 arg2 ";ped file1:symb"↵
```

Redirection of standard I/O

The default files for standard input and output are the keyboard and the terminal. When starting a C program you can give redirection commands to define other files for standard input and output.

REDIRECTION COMMAND:



A "<" (less than) character indicates that input shall be taken from the file whose name is specified after the "<" character. If the file does not exist, an error message is given. A ">" (greater than) character causes output to be written to the file whose name is specified after the redirection character. A ">>" sign causes the output to be appended to the file whose name is given. If the output file does not exist it will be created.

The characters "<" and ">" are only interpreted as redirection signs, if they are not embedded between apostrophes.

Default file type for input and output file is :symb.

example

```
@nd-500 myprog <infile:symb arg1 arg2 > outfile:list↵
```

The program myprog is called with the parameters arg1 and arg2. Standard input is taken from the file infile:symb whereas output is written to outfile:list.

Parameter files

The length of the SINTRAN command line is limited to 103 characters (which can be different in following SINTRAN versions). Since C programs may need argument lists which exceed this length it is possible to write the argument list to a file. The name of this file is part of the command line. It is introduced by a "@". This implies that a "@" can not be passed as the first character of a parameter, it can be passed on any other position if it is embedded between apostrophes. The maximum number of characters in a parameter file is 2000 (the size of the internal command line buffer).

Line delimiters in the parameter file are treated as blanks.

example

```
@nd myprog arg1 @para-file:symb arg2↵
```

The program myprog is started with two arguments (arg1 and arg2) and a parameter file (para-file:symb).

upper/lower case letters

SINTRAN converts all characters of the command line to upper case characters. As C is rather based on lower case characters the command line is internally converted to lower case letters. A character (a..z) is converted to an uppercase one by a leading "^" . If you need a "^" in a parameter you have to write "^^" .

The conversion to lower case letters does not apply to continuation lines.

example

```
@nd-500 myprog ^This Shows, how "character ^^"\↵
>for UPPER case ^l^e^t^t^e^r^s "is treated"↵
```

These command lines will call program "myprog" and pass the following 10 arguments:

0. argument: "SOURCE-OF-MYPROG"
(if the source file name is: "SOURCE-OF-MYPROG:C")
1. argument: "This"
2. argument: "shows,"
3. argument: "how"
4. argument: "character ^"
5. argument: "for"
6. argument: "UPPER"
7. argument: "case"
8. argument: "LETTERS"
9. argument: "is treated"

"'"

SINTRAN ignores a "'" character and all following characters in a command line. If you want to pass a "'" in a parameter you either have to write it in a continuation line which is no longer a SINTRAN command line or have to specify it as octal number ("\047"). In both cases it must be embedded in apostrophes and be escaped by a "\".

The two arguments "It's not nice" and "that you can't" can be passed to the C program myprog in the following way:

```
@nd myprog " It\047s not nice"\↲
>"that you can\'t""↲
```

Program parameters

If you want to pass arguments from the command line to a program the function main must be declared with two parameters usually called "argc" and "argv".

declaration

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

"Argc" contains the number of arguments being passed, "argv" is a pointer to an array of strings which contain the arguments. All parameters are passed as strings.

The first argument argv[0] contains the name of the source file, argv[1] the first parameter, argv[2] the second parameter etc..

parameter syntax

Parameters have to be separated by blanks. If you want to pass a blank within a parameter you have to enclose the parameter by quotation marks (" ").

special characters

All parameters containing the following special characters have to be enclosed by quotation marks:

"\b" back space
"\t" horizontal tab
"\n" new line
"\f" form feed (line feed)
"\ddd" octal number

escape character

If a parameter shall contain one of the characters "\", """ or "'" the character has to be escaped by a backslash ("\\", "\"", "\'"). Special conditions how to pass an apostrophe you find above.

< >

Parameters containing a "less than" "<" or "greater than" ">" character must be enclosed by quotation marks. Otherwise input/output is redirected.

@ ; "@" and ";" can not be passed as first character in an argument. If they occur on another position the parameter must be enclosed by quotation marks.

^ If a "^" character shall be part of an argument it has to be written as "^^". Otherwise it is interpreted as "convert next character to uppercase".

You will find the program "prog-to-exec" on page 13-29. It interprets the command line and lists the arguments on the terminal. So you can try out different argument lists and see what happens.

Chapter 13

C Library Functions

General

Under SINTRAN, the functions described in this chapter are all stored in the C library. However, to emphasise the UNIX function levels, we divided the functions into two sections. The section "Basic Functions" describes lower level functions, which, under UNIX, are calls to the operating system. The section "Standard Functions" describes higher level functions, which represent the original part of the C runtime system.

Header Files

Header files contain macro definitions and function declarations, which are used by the C library functions. Declarations for related functions are grouped in a common header file, which you must **include** in your program when calling one of these functions. The include commands have to be given at the top of your program. Functions with **int** or **void** as result type may be used without including any header file. Which header file you have to include in order to use a certain function will be specified in the individual function description. Below you will find a list of all header files with a short description of their contents:

errno.h	macro definitions of error constants used by the C runtime system
stdio.h	I/O macro definitions and function declarations
ctype.h	declarations of character functions
math.h	declarations of external mathematical functions
fcntl.h	macro definitions used by the function *open*
stat.h	macro definitions used by the functions *open*, *stat*, *lstat* and *fstat*; prior to stat.h you have to include types.h (to make some type definitions known) and time.h
setjmp.h	declarations of functions that handle global jumps

memory.h	declarations of memory functions
string.h	declarations of string functions
time.h *times.h* *timeb.h* *types.h*	declarations used by time functions
varargs.h	macro definitions for using variable argument lists (see description of varargs on page 13-101).

Standard Files

There are four standard files, which are implicitly open:

- The standard input file always has the file number 0 and is associated with the SINTRAN standard input device, which is usually your terminal (if you are in an interactive process).

- The standard output file always has the file number 1 and is associated with the SINTRAN standard output device, which is usually your terminal (if you are in an interactive process). This file is always line-buffered.

- The standard error file always has the file number 2 and is associated with the SINTRAN error device, which is usually your terminal (if you are in an interactive process). It is used by the runtime system for error messages. This file is always unbuffered, i.e. error messages are sent to your screen as soon as they are written.

- The standard temporary file is the SINTRAN standard scratch file (SCRATCH)SCRATCHXX:DATA with the file number 64 (octal: 100). To this file you have read, write and append access.

File Names

The C runtime system accepts all file names (*path names* in UNIX terms) that conform to the SINTRAN naming conventions and which are described in the chapter "The File System" of the Time Sharing and Batch Guide. The default file type is *:SYMB*.

For compatibility reasons UNIX file names are accepted as well. If the UNIX file name does not contain more than one directory specification, it is converted to SINTRAN notation according to the following rules:

- Leading dots (. or ..) will be ignored.
- Directory names, which are separated by slashes (/), are converted into SINTRAN user names.
- Dots within the name are replaced by dashes (-).
- The last dot is converted into a colon (:), so that the last part of the file name is taken as the SINTRAN file type.
- All other characters of an UNIX file name remain unchanged.

If an UNIX file name cannot be converted or the converted file name still contains invalid characters, SINTRAN system calls will report an error.

examples

UNIX notation		SINTRAN notation
../user/file.name	→	(user)file:name
sys/myfile.h	→	(sys)myfile:h
/user/a.b.c.list	→	(user)a-b-c:list

Notation

The functions are documented in the following way:

FUNCTION

The heading FUNCTION gives the name of the function described. Similar functions are described together.

HEADER FILE

This heading specifies the header file you have to include in order to use the function. If no header file is given, you can use the function without including a header file.

DECLARATION	The heading DECLARATION specifies the declarations of the function and its parameters as they appear in the C library. You need not declare any function in your program, but only include the header file specified in the individual description, if any.
DESCRIPTION	Under this heading the function is described.
RETURN VALUE	The heading RETURN VALUE specifies the type of the function result and describes the values that can be expected.
NOTES	The heading NOTES explains special precautions and particularities of the function described.

Error Handling

For a proper error handling you should always include the header file *errno.h* in your program:

```
#include <errno.h>
```

This file contains the declarations of two integer variables *errno* and *OSerrno*. On an unsuccessful function call *errno* contains an error number of the C runtime system which describes the error situation, whereas in *OSerrno* the SINTRAN error code is made available (see SINTRAN III Reference Manual). In general, *OSerrno* gives a more detailed description of the error. If no corresponding SINTRAN error code exists, *OSerrno* is set to zero.

errno

OSerrno

> **Note**
> After a successful call *errno* is not cleared. So, error numbers should only be tested after an error has been indicated by the return value of the function called.

Below you will find a list of all *errno* values used in this implementation together with their macro names, as defined in *errno.h*, and their meaning. (In order to be compatible to UNIX, the header file also defines some error constants which are not used under SINTRAN.)

2 ENOENT	No such file, user, directory
5 EIO	Error in I/O operation
6 ENXIO	Hardware error in I/O operation
7 E2BIG	Parameter list too long
9 EBADF	Wrong file number
13 EACCES	Access permission denied
14 EFAULT	Illegal address in system call
16 EBUSY	File or directory in use
17 EEXIST	File already exists
22 EINVAL	Invalid parameter
23 ENFILE	SINTRAN file buffer overflow

24 EMFILE	Attempt to open too many files
28 ENOSPC	No more space available
33 EDOM	Illegal parameter to mathematical function
34 ERANGE	Illegal result of mathematical function: The result cannot be represented within machine precision, e.g. overflow.
60 ETIMEDOUT	Timeout while accessing a remote system
61 ECONNREFUSED	No connection to remote system

FUNCTION print error message: *perror*

HEADER FILE #include <errno.h>

DECLARATION

```
● void perror (s);
  char *s;
```

DESCRIPTION

perror produces a message on the standard error output device (which is the terminal in most cases), describing the last error encountered during a function call.

The user specified string *s* is displayed first, followed by a colon, a blank, and then the *errno* message and a carriage return. Most usefully, the string parameter is the name of the program part in which the error occurred.

If *OSerrno* is unequal zero, the *errno* message will be followed by a slash and the SINTRAN error number.

To enable you to access the standard error messages the following two variables are declared in the header file *errno.h*:

sys_nerr **int** sys_nerr;
sys_errlist **char** *sys_errlist[];

sys_errlist is an array containing the error messages. The error variable *errno* can be used as subscript. *sys_nerr* is the number of entries in the table.

example

```
#include <stdio.h>

main()
{
  FILE *fp;

  fp = fopen ("the-door", "r"); /* This file      */
                                /* does not exist */

  if (fp == NULL) {
    perror ("Program TEST-PERROR");
    clearerr(fp);
  }
}
```

The following error message will be displayed:

Program TEST-PERROR: No such file, user or directory
SINTRAN error = 46

Basic Functions

Basic I/O

The functions described in this section are also used internally by the functions of the formatted I/O package (see page 13-39). In order to avoid problems, you should not intermix functions of these two sections. The chart in appendix C on page 17-81 shows the relationship between basic and formatted I/O.

As the system buffers the data it is possible that you get a delayed error message after an error occured, i.e. although a call to *write* is erroneous you can get the error message later with a call to *fsync* or *close*.

FUNCTION

open a file: *open*

HEADER FILE

```
#include <fcntl.h>
```

DECLARATION

```
• int open (fname, flags, mode);
  char *fname;
  int flags, mode;
```

DESCRIPTION

The function *open* connects the physical file *fname* to your program with the access rights specified in *flags*. On a successful call it returns a positive file number, which identifies a file descriptor containing the current file position. Initially the file position is set to zero; it is updated by *read*, *write* and *lseek* (see pages 13-14, 13-16 and 13-21). The file number returned by *open* will be used as a reference when accessing or manipulating the file. Under SINTRAN for each program a table of allocated file numbers is maintained. The size of this table can be determined by calling the function *getdtablesize* (see page 13-19).

fname

A file name written in UNIX notation is automatically transformed into a SINTRAN file name. If the file name is an empty string (""), you are expected to specify a SINTRAN device number as third parameter.

mode

The parameter *mode* specifies the access rights with which a file is to be created.

mode == 0	If the *mode* equals zero and if a creation mask is defined by the function *umask* (see page 13-20), this mask is used to set the access rights. Otherwise, if the mode equals zero and no creation mask is defined, the user's SINTRAN defaults for creation are taken. You can list your defaults with the SINTRAN command USER-STATISTICS.
mode != 0;	If *mode* is unequal zero, it is used as creation mode mask. In the header file *stat.h* integer macros for *mode* are defined. If you include this header file, you can use them to specify individual access rights for a file. The following values are defined:

S_IREAD	own read access
S_IWRITE	own write access
S_GREAD	friend read access
S_GWRITE	friend write access
S_PREAD	public read access
S_PWRITE	public write access

You can combine them by using the bitwise OR operator, e.g. S_READ | S_IWRITE | S_GREAD.

flags

The parameter *flags* specifies the access rights with which a file is to be opened. For this purpose the following macros are defined:

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.
O_NDELAY	Immediate return, if a file is blocked
O_APPEND	Each write appends at the end of the file.
O_CREAT	If the file does not exist, it will be created and opened. The file name must not be abbreviated. The default file type is :SYMB.
O_TRUNC	If the file exists, its length is truncated to 0; a write operation immediately following *open* will start at file offset 0.
O_EXCL	Only exclusive access is allowed. An error occurs, if you try to create an already existing file.
O_S3NABBR	SINTRAN extension: When opening an existing file an exact match of user-specified and SINTRAN file name is required.
O_S3CHAR	SINTRAN extension: Except terminals all files are considered to contain binary data by default. On read operations this flag causes a file to be considered as a stream of characters and the parity bit is removed.
O_S3COM	SINTRAN extension: If the file had been created with the Monitor call "CreateFile" as a contiguous file (in standard C there is no possibility to set the number of pages for the file to be created) it will be opened for common access, otherwise the flag will be

ignored. If the file does not exist and O_CREAT is set, the setting of O_S3COM does neither cause the file to be created as contiguous file nor will it be opened for common access.

O_S3SEG SINTRAN extension: If possible, the file will be connected to a segment, which enables a faster random access. (Segments are described in the ND-500 Loader/Monitor manual.)

You will find an example of *open* on page 13-37.

Again, these flags may be combined by using the bitwise OR operator, e.g. O_WRONLY | O_CREAT.

RETURN VALUE

If the call was successful, the SINTRAN file number will be returned. On error the value -1 will be returned and the error variable *errno* is set to one of the following values:

ENOENT	O_CREAT is not set and the file does not exist.
EACCES	Access to the file denied because of missing access rights.
EMFILE	No more file numbers available.
ENXIO	The file is linked to a device without hardware access.
EFAULT	The pointer to *fname* is outside the address space.
EEXIST	O_EXCL and O_CREAT was specified, but the file exists.

NOTES

A program may at most have 64 files opened simultaneously.

FUNCTION

get segment number: segment number

DECLARATION

```
int segment_number(fd);
int fd;
```

DESCRIPTION

The routine segment_number returns the segment number of the file, which is opened as segment and associated with the file number fd. If fd refers to a file not opened as segment -1 is returned. If fd is not associated with an open file at all errno is set to EBADF.

FUNCTION

create a file: *creat*

DECLARATION

- **int** creat (fname, mode);
  char *fname;
  int mode;

DESCRIPTION

creat creates a new file or prepares to rewrite an existing file called *fname*. It is implemented as a call to *open* with the following parameters:

open (fname, O_RDWR|O_CREAT|O_TRUNC|O_S3CHAR, mode)

RETURN VALUE

The return values and the setting of *errno* are the same as for *open*.

FUNCTION

read from a file: *read*

DECLARATION

```
• int read (fn, buf, nbyte);
  int fn;
  char *buf;
  unsigned nbyte;
```

DESCRIPTION

The function *read* reads *nbyte* bytes from the file associated with *fn* into the buffer pointed to by *buf*. The file number *fn* is obtained from a previous *open* or *creat* call. After each read the position of the internal file pointer is incremented by the number of bytes read.

When reading from the terminal, the following has to be considered:

The input will be line buffered which means that for each line of input *read* must be called.

When not reading characterwise (nbyte > 1) the following characters are control characters:

CTRL+@	end of input (will not be written to buf)
<newline>	end of input (only the carriage return = OXOD will be written to buf)
CTRL+A	remove previous character from input line
CTRL+K	clear current input line
CTRL+R	rewrite the line as it looks now (does not affect buf)

When reading characterwise from the terminal (nbyte == 1) the characters mentioned above do not control the input. There has to be a programmed end-of-input condition. See example on page 13-34.

RETURN VALUE

After successful execution, the number of bytes actually read and placed in the buffer is returned. The number of bytes returned is less than specified, if the end of file was encountered, or if the input from a line-oriented device (e.g. a terminal) is terminated by a carriage return. The return value 0 indicates that the end of the file is reached, or that *nbytes* was less than or equal to zero.

On error -1 is returned and *errno* is set to one of the following values:

EBADF	*fn* is not associated with an open file.
EACCES	*fn* is not associated with a file opened for reading.
EFAULT	*buf* points to an invalid address.

NOTES

Under SINTRAN most character files contain parity bits. If you want the parity bits to be removed, the file must have been opened with the O_S3CHAR flag specified.

You will find an example of *read* on page 13-34.

FUNCTION write to a file: *write*

DECLARATION

```
• int write (fn, buf, nbyte);
  int fn;
  char *buf;
  unsigned nbyte;
```

DESCRIPTION

The function *write* writes *nbyte* bytes from the buffer pointed to by *buf* to the file associated with *fn*. After each write the position of the internal file pointer is incremented by the number of bytes written. If the file was opened with O_APPEND, the file pointer is set to the end of the file before the first write access.

If you have filled a buffer with input from a terminal all characters after the carriage return (OXOD) are truncated. So if you send the buffer to the terminal you have to add a desired line-feed character (OXOA).

RETURN VALUE

After successful execution the number of bytes actually written is returned. Otherwise, -1 is returned and *errno* is set to one of the following values:

EBADF	*fn* is not associated with an open file
EACCES	*fn* is not associated with a file opened for writing.
EFAULT	*buf* points to an invalid address.
EIO	hardware error

You will find an example of *write* on page 13-34.

FUNCTION

flush buffers of the basic I/O system: *sync*, *fsync*

DECLARATION

- **int** sync();
- **int** fsync (fn);
  int fn;

DESCRIPTION

fsync writes the buffered data of the file associated with *fn* to its permanent storage device, while *sync* flushes all buffered data of the basic I/O system.

Both, *fsync* and *sync* set the SINTRAN file pointer to the end of the file.

RETURN VALUE

The return value 0 indicates a successful call. On error -1 is returned and *errno* is set to one of the following values:

EBADF	*fn* does not refer to an open file.
EIO	error in I/O operation

NOTES

fsync and *sync* only flush buffers of the basic I/O system. To flush buffers of the formatted I/O system you have to use the function *fflush* (see page 13-46).

You will find an example of *fsync* on page 13-34

FUNCTION close a file: *close*

DECLARATION

- **int** close (fn);
  int fn;

DESCRIPTION *close* closes the file associated with the file number *fn*. This function is called automatically when terminating the program.

RETURN VALUE After successful execution 0 is returned. Otherwise, -1 is returned and *errno* is set to EBADF, indicating that *fn* is not associated with an open file.

You will find an example of *close* on page 13-34.

FUNCTION delete a file: *unlink*

DECLARATION

- **int** unlink (fname);
  char *fname;

DESCRIPTION *unlink* deletes the file specified in *fname*. The file name may be abbreviated. *unlink* may only be applied to closed files.

RETURN VALUE After successful execution 0 is returned. Otherwise, -1 is returned and *errno* is set to one of the following values to indicate the error:

ENOENT	The file specified does not exist, or there are more than one file names with the given abbreviation, or wrong syntax of file name.
EACCES	You do not have the access right to delete the file.
EBUSY	The file is still open.
EFAULT	Illegal address in system call, e.g. *fname* equals zero.

You will find an example of unlink on page 13-34.

FUNCTION — get file number table size: *getdtablesize*

DECLARATION

- int getdtablesize ();

DESCRIPTION

The file number table contains an entry for each file opened by *open*. These entries are numbered with integers, starting at 0 and incrementing by 1 for each new entry. The maximum size of this table can be obtained by calling *getdtablesize*.

RETURN VALUE

getdtablesize returns the maximum size of the file number table.

FUNCTION — duplicate file number: *dup, dup2*

DECLARATION

- ```
  int dup (old_fn);
  int old_fn;
  ```
- ```
  int dup2 (old_fn, new_fn);
  int old_fn, new_fn;
  ```

DESCRIPTION

dup

The function *dup* returns a new file number which refers to the same file descriptor as the file number given as parameter.

dup2

The function *dup2* works like *dup* the only difference being that the new file number *new_fn* is supplied by the user. The value of *new_fn* must be in the range of 0..getdtablesize-1 (see page 13-19). If *new_fn* is an already active file number, the file currently referred to is closed before. A typical application of *dup2* is redirection of standard input and output.

RETURN VALUE

The return value -1 indicates an error and *errno* is set to one of the following values:

EBADF	*old_fn* or *new_fn* is not a valid file number.
EMFILE	There are no more file numbers available.

FUNCTION set file creation mode mask: *umask*

DECLARATION

```
• int umask (mode);
  int mode;
```

DESCRIPTION The function *umask* sets the default access rights for files that have to be created. The *mode* can be defined like the *mode* parameter in *open*, using the flags defined in the header file *stat.h* (see page 13-11).

RETURN VALUE As result the previous creation mask is returned.

You will find an example of *umask* on page 13-33.

FUNCTION change mode of file: *chmod*

DECLARATION

```
• int chmod (fname, mode);
  char *fname;
  int mode;
```

DESCRIPTION *chmod* changes the access rights of the file *fname* according to the mask defined by *mode*. The mask can be defined by using the flags of the header file *stat.h* (see also page 13-11). When calling *chmod* the file must not be open.

RETURN VALUE After successful execution 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

You will find an example of *chmod* on page 13-38.

FUNCTION determine accessibility of a file: *access*

DECLARATION

```
● int access(fname, mode);
  char *fname;
  int mode;
```

DESCRIPTION

The function *access* determines whether the file specified by *fname* exists and whether it can be accessed at least for reading. The parameter *mode* is ignored.

RETURN VALUE

If the file exists and can be accessed, 0 is returned. Otherwise, -1 will be returned and *errno* is set to indicate the error.

FUNCTION reposition a file pointer: *lseek*

DECLARATION

```
● long lseek (fn, offset, position);
  int fn;
  long offset;
  int position;
```

DESCRIPTION

lseek sets the file pointer of the file associated with *fn* *offset* bytes from the beginning (position), from the current position (*position=1*) or from the end of the file (*position=2*). For *offset* negative values may be specified as well as positive ones.

RETURN VALUE

After successful execution the new file position (measured in bytes from the beginning) is returned. Otherwise, -1 is returned and *errno* is set to one of the following values:

EBADF	*fn* is not associated with an open file.
EINVAL	The *position* specified was no valid value (0, 1 or 2), or the new file pointer position would be negative.

You will find an example of *lseek* on page 13-34.

FUNCTION

truncate a file: *truncate*, *ftruncate*

DECLARATION

- ```
  int truncate (fname, length);
  char *fname;
  int length;
  ```
- ```
  int ftruncate (fn, length);
  int fn, length;
  ```

DESCRIPTION

truncate

truncate sets the size of the file *fname* to *length* bytes. If the file previously was larger, the extra data gets lost. Note, that the file must not be open!

ftruncate

ftruncate performs the same as *truncate* but on an open file. Instead of the file name, the file number retrieved from a previous *open* call is specified.

RETURN VALUE

After successful execution 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

You will find an example of *truncate* on page 13-37.

FUNCTION — get file status: *stat, lstat, fstat*

HEADER FILE

```
#include <types.h>
#include <stat.h>
```

DECLARATION

```
  int stat (fname, buf);
  char *fname;
• struct stat *buf;

• int lstat (fname, buf);
  char *fname;
  struct stat *buf;

• int fstat (fn, buf);
  int fn;
  struct stat *buf;
```

DESCRIPTION

All three functions fill the structure *stat* pointed to by *buf*. This structure contains information about the specified file and is declared in the header file *stat.h* as:

```
struct stat {
      short          st_dev;       /* directory index the file resides on    */
      unsigned long  st_ino;       /* object index                           */
      unsigned short st_mode;      /* access protection bits                 */
      unsigned short st_s3mode;    /* SINTRAN access protection bits         */
      short          st_nlink;     /* under SINTRAN always 1                 */
      short          st_uid;       /* user index                             */
      short          st_gid;       /* user index                             */
      short          st_rdev;      /* is unequal zero if peripheral device   */
      int            st_size;      /* number of bytes in file                */
      long           st_atime;     /* time of last read access in seconds    */
      int            st_spare1;
      long           st_mtime;     /* time of last write access in seconds   */
      int            st_spare2;
      long           st_ctime;     /* time of creation in seconds            */
      int            st_spare3;
      long           st_blksize;   /* optimal block size for file I/O (2048) */
      long           st_blocks;    /* number of blocks allocated             */
      long           st_spare4[2];
      }
```

The time values returned in the structure can be decoded through the time function *ctime*, *localtime* or *gmtime* (see page 13-37).

Since there are no symbolic links in the SINTRAN file system, the functions *stat* and *lstat* are identical.

The function *fstat* delivers the same result as *stat* and *lstat*, but instead of a file name it requires a file number *fn* from a previous call to *open* as first parameter.

RETURN VALUE

After successful execution 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

NOTES

These functions are implemented as close to the corresponding UNIX functions as possible.

FUNCTION

set file times: *utimes*

HEADER FILE

```
#include <time.h>
```

DECLARATION

```
• int utimes (fname, tv);
  char *fname;
  struct timeval *tv[2];
```

DESCRIPTION

The function *utimes* sets the dates of the last read and write access in the descriptor of the file *fname* to the time values specified by *tv*. The structure *timeval* is declared in the header file *time.h* as:

```
struct timeval {
        long tv_sec;
         long tv_usec;
        };
```

RETURN VALUE

After successful execution 0 is returned. Otherwise -1 is returned and errno is set to indicate the error.

You will find an example of *utimes* on page 13-38.

FUNCTION rename a file: *rename*

DECLARATION

```
• int rename(old_name, new_name);
  char *old_name, *new_name;
```

DESCRIPTION

The function *rename* changes a SINTRAN file name from *old_name* to *new_name*. If no file type is specified, the file will get the default type :SYMB. The contents of the file remains unchanged.

RETURN VALUE

After successful execution 0 is returned. Otherwise, -1 is returned and *errno* is set to one of the following values:

ENOENT	The specified old file does not exist, or the file name abbreviation was ambiguous, or the syntax of the old file name is wrong.
EACCES	You are not authorised to change the file name.
EBUSY	The old file is still open.
EEXIST	A file with the specified new file name exists already.
EFAULT	Illegal address in system call.

You will find an example of *rename* on page 13-38.

FUNCTION set echo strategy: echomode, echo_mode

DECLARATION

```
• void echomode(mode);
  int mode;
  void echo_mode(ldn, mode);
  int ldn, mode;
```

DESCRIPTION

Echomode sets an echo strategy for the terminal whereas echo_mode sets an echo strategy for an open file. The parameter ldn contains the number of the open file which is either found out by a call to *open* or with formatted I/O by a call to *fileno(iop)* (declaration: FILE *iop;). The input parameter mode determines one of the following strategies:

```
mode < 0 : no echo
mode = 0 : echo on all characters
mode = 1 : echo on all except control characters.
```

You will find an example of *echo_mode* on page 13-113.

FUNCTION set break strategy: breakmode, break_mode

DECLARATION

```
• void breakmode(mode);
  int mode;
  void break_mode(ldn, mode);
  int ldn, mode;
```

DESCRIPTION Breakmode sets a break strategy for the terminal whereas break_mode sets the break strategy for an open file. The parameter ldn contains the number of the open file which is either found out by a call to *open* or with formatted I/O by a call to *fileno(iop)* (declaration: FILE *iop;). The input parameter mode determines one of the following strategies: mode < 0 : no break mode = 0 : break on all characters mode = 1 : break only on control characters.

You will find an example of *break_mode* on page 13-113.

— Other Basic Functions —

FUNCTION execute a program: *execve*

DECLARATION

- int execve (name, argv, envp);
  char *name, *argv[], *envp[];

DESCRIPTION

The function *execve* executes the program specified by *name*. If name specifies an ND-500 program which is not defined as standard domain, it must start with "ND-500" followed by a blank. Execve builds a string containing the name and the parameter list. Missing quotation marks which should enclose parameters containing blanks are added. The resulting string is passed to the operating system to be executed. To SINTRAN as well as to the C runtime system this string corresponds to the command line. The rules described in chapter "The command line" on page 12-6 are valid.

For "special characters" care must be taken to pass them correctly (see chapter "The command line"). If, for example, a backslash not enclosed in quotation marks is the last argument or the last character of the last argument, you are prompted for more arguments on a continuation line (with a ">" character). If the arguments are read from the terminal no "carriage return" may be part of an argument as SINTRAN would interpret this as "end of command line".

To avoid the SINTRAN error "TOO LONG STRING" at an attempt to pass more than 103 characters *execve* returns -1 and "errno" is set to "E2BIG".

argv

The array *argv* contains the optional parameter list starting with argv[1];. Argv[0] is not transferred. The program being called deliveres it's source file name as argv[0] subsequently.

Envp is a dummy parameter which may be left out for a SINTRAN program. For portability reasons at least a (null) pointer to envp should be defined.

Before execve is executed and the control is given to SINTRAN all files except the standard files are closed.

A successful *execve* can never return to the calling program, because the calling core image gets lost.

RETURN VALUE

In case of an error, the value -1 will be returned. If the parameter list was too long, *errno* is set to

E2BIG otherwise *name* did not specify a valid program.

example

```
execve ("ND-500 my-domain", parmlist);
execve ("PED", NULL);
```

```
/* This is the source of a program that calls the basic library   */
/* function execve.                                                */
/* It reads strings from the terminal and interprets them as      */
/* arguments to be passed to the other program called via execve. */

extern execve();
#include <stdio.h>
#include <errno.h>

main()
{
 char *out1 = "Argument: ";
 char CR = 0X0D;
 char *prog_name="nd-500 execute";
 char *argv[11], argument[11][80];
 int i, j, k, sin_strlen;

 printf("A program to show how execve works\n\n");
 printf("Input of arguments to be passed to the program, \n");
 printf(<being executed at the end with a call of execve:\n");
 printf("(No more than 10 arguments are accepted)\n");
 printf("END WITH <CR> AS ARGUMENT\n");
 argv[0] = prog_name;       /* dummy parameter which isn't passed */
 sin_strlen = strlen(prog_name) + 1;
 i = 0;
 do {
  i++;
  write(fileno(stdout),out1,strlen(out1));
  j = read(fileno(stdin),&argument[i][0],80);
  if (argument[i][0] != CR) {
   argument[i][j-1] = '\0';
   k = 0;
   while ((argument[i][k] != ' ') && (k < j-1)) k++;
   if (k < j-1) sin_strlen = sin_strlen + 2;
   argv[i] = argument[i];
   sin_strlen = sin_strlen + j;
  }
  else j = 0;
 } while ((i < 10) && (j > 0));
 if ((i == 10) && j) {
  printf("Maximum number of arguments(%d) was given\n",i);
  argv[i+1] = 0;
 }
 if (j == 0) {
  printf("%d arguments were given\n", i-1);
  argv[i] = 0;
 }
```

```
 printf("The string for SINTRAN will be of length %d\n",sin_strlen);
 if (sin_strlen >= 103)
  printf("Your parameter list is too long for SINTRAN!\nAnyway. ");
 printf("\"execve\" will be called now!\n");
 i = execve(prog_name, argv);
 if (i == -1) {
  printf("ERROR execve: ");
  if (errno == E2BIG)
   printf("parameter list too long!!!\n");
  else
   printf("\"%s\" isn't a valid program name!!!\n",prog_name);
 }
}


/* This is the source of the program which will be called from */
/* the program whose source text is written above.             */
/* It has to be linked to a ND-500 domain named "execute" !!!  */
/*                                                             */
/* It simply prints out the passed arguments.                  */

#include <stdio.h>

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

 printf("\nThis is now the program called via execve\n");
 printf("argc = %d\n",argc);
 printf("The passed arguments are:\n");
 for (i=0; i<argc; i++) printf("%d. argument:>%s<\n",i,argv[i]);
}
```

FUNCTION get time in seconds: *gettimeofday*

HEADER FILE

```
#include <time.h>
```

DECLARATION

```
• int gettimeofday (tp, tzp);
  struct timeval *tp;
  struct timezone *tzp;
```

DESCRIPTION

The function *gettimeofday* fills in the structures *timeval* and *timezone*, which are declared in the header file *time.h* as follows:

timeval

```
struct timeval {
        long tv_sec;
        long tv_usec;
        };
```

timezone

```
struct timezone {
        int tz_minuteswest;
        int tz_dsttime;
        };
```

After the call *tv_sec* will contain the time in seconds since 1st January 1970, 00:00:00 hrs and *tv_usec* will contain the microseconds of the current second.

As under SINTRAN the time zone is always considered to be Greenwich, the values of the structure *timezone* are

set to zero. Therefore, when calling *gettimeofday tzp* may be a NULL pointer.

RETURN VALUE

After successful execution 0 is returned, otherwise -1 is returned and *errno* is set to EINVAL indicating that *tp* is an invalid pointer.

example

```
#include <time.h>
#include <stdio.h>

main()
{
  struct timeval tp;
  gettimeofday (&tp,NULL);
  printf ("tv_sec  = %d\n",tp.tv_sec);
  printf ("tv_usec = %d\n",tp.tv_usec);
}
```

The output could look like:

```
tv_sec  = 538309537
tv_usec = 54120000
```

NOTES

gettimeofday is almost identical to the standard function *ftime*, with the only difference that *gettimeofday* returns microseconds and not milliseconds.

FUNCTION

get system page size: *getpagesize*

DECLARATION

- int getpagesize();

DESCRIPTION

getpagesize returns the page size used for swapping and file system I/O. Currently this size is 2048 bytes.

FUNCTION

get user/process identification: *getuid*, *getpid*

DECLARATION

- int getuid();
- int getpid();

DESCRIPTION

Getuid returns a unique number for the current process. For interactive and batch processes this is the logical device number, i.e. the terminal number; for RT-processes the RT description address is returned. *Getpid* returns the RT description adress.

example

```
main()
{
  printf("User %s logged in ", getlogin());
  printf("on terminal %d \n", getuid());
  printf("Process number %d \n", getpid());
}
```

Examples of BASIC-I/O

```
/* This program reads input from the terminal (up to a programmed     */
/* end) into a buffer, writes this buffer to a file opened with       */
/* "creat", reads the file (from the beginning) to a buffer and       */
/* displays the contents of the buffer on the terminal again.         */
/*                                                                    */
/* Care must be taken when reading from/writing to a terminal:        */
/*   If reading is done with length 1, a programmed end-condition     */
/*   has to be there and the special control characters don't work.  */
/*   If reading is done with a normal buffer length, input does not   */
/*   only end with CRTL+@, but also with CR (line buffered). The      */
/*   special control characters are available.                        */
/*   If CR+LF are read from the terminal, only the CR is buffered,    */
/*   so if you want to output the terminal input on the terminal      */
/*   again with the occurrence of CR+LF you has to write an           */
/*   additional LF.                                                   */
/* Calls of: umask, creat, read, write, fsync, close, unlink, lseek,  */
/*          perror, fileno.                                           */
#include <stdio.h>
#include <fcntl.h>
#include <types.h>
#include <stat.h>
#define BUFLEN 2048
main()
{
 char new_line[2];
 char *out1 = "Type what you would like to be written to file1!";
 char *out2 = "(End your input with a double $-sign: $$)";
 char *out3 = "Your input, read from file1:";
 char *out_err      = "That was no input!";
 char buffer[BUFLEN];
 int fn1, fmask;
 int i, j;
 new_line[0] = 0X0D;
 new_line[1] = 0X0A;
             /* set default file access: own read and write access, */
             /* read access for friends and others.                 */
 fmask = umask(S_IREAD | S_IWRITE | S_GREAD | S_PREAD);
             /* Create file "file1:symb" with default file access.  */
             /* The file will be opened for read/write.             */
 if ((fn1=creat("file1:symb", S_IREAD | S_IWRITE |
                              S_GREAD | S_PREAD)) < 0) {
                               /* on error: print the error and exit */
  perror("Creating file1:symb");
  exit(1);
 }
                          /* write leading text to the terminal     */
                          /* (new lines must be written explicitly) */
 write(fileno(stdout), out1, strlen(out1));
 write(fileno(stdout), &new_line[0], 2);
 write(fileno(stdout), out2, strlen(out2));
 write(fileno(stdout), &new_line[0], 2);
```

```
     /* as the end of input condition consists of two characters    */
     /* - one $ may occur as normal input - one byte is read before */
     /* the read-loop starts:                                       */
i = 0;
read(fileno(stdin), &buffer[i], 1);
                     /* read 1 byte from the terminal to the buffer */
                     /* as long as no two subsequent $ occur        */
do {
 i++;
 read(fileno(stdin), &buffer[i], 1);
}
while (!(buffer[i] == '$' && buffer[i-1] == '$'));
i = i - 2;                  /* forget about the two $ in the buffer */
           /* write buffer (with length of terminal input) to file */
if (i >= 0) j = write(fn1, &buffer[0], i+1);
else {     /* no input: write a message, close and delete file1    */
 write(fileno(stdout), &new_line[0], 2);
 write(fileno(stdout), out_err, strlen(out_err));
 close(fn1);
 unlink("file1:symb");
 exit(1);
}
fsync(fn1);     /* write the c-buffered data to the SINTRAN buffer  */
lseek(fn1,0,0); /* set the filepointer to the beginning of the file */
for (i=0; i<BUFLEN-1; i++) buffer[i] = ' ';        /* clear buffer */
                    /* read from the file into the internal buffer  */
j = read(fn1, &buffer[0], BUFLEN);
/* write leading text to the terminal */
write(fileno(stdout), &new_line[0], 2);
write(fileno(stdout), out3, strlen(out3));
write(fileno(stdout), &new_line[0], 2);
/* write the internal buffer to the terminal as long as it contains */
/* characters read from the file. If an CR occurs in the buffer,    */
                                                /* add a line feed */
for (i=0; i<j; i++) {
 write(fileno(stdout), &buffer[i], 1);
 if (buffer[i] == 0X0D) write(fileno(stdout), &new_line[1], 1);
}
/* close and delete file file1:symb, associated with file no. fn1   */
close(fn1);
unlink("file1:symb");
}
```

BASIC-IO-2

```
/* This program reads input from the terminal into a buffer, (up      */
/* to an input of CRTL+@) writes this buffer to a file opened with    */
/* "creat", reads the file (from the beginning) into a buffer and     */
/* redisplays the contents of the buffer on the terminal.             */
/* The program is functionally similar to the preceding one but       */
/* terminal input is read linebuffered, in order to make the special  */
/* control charcters work.                                            */
/* Calls of: umask, creat, read, write, fsync, close, unlink, lseek,  */
/*           perror, fileno.                                          */
```

```
#include <stdio.h>
#include <fcntl.h>
#include <types.h>
#include <stat.h>
#define BUFLEN 2048
main()
{
 char new_line[2];
 char *out1 = "Type what you would like to be written to file1!";
 char *out2 = "(End of input: CTRL+@)";
 char *out3 = "Your input, read from file1:";
 char *out_err = "That was no input!";
 char buffer[BUFLEN];
 int fn1, fmask;
 int b_written, b_read, bufpos, i;
 new_line[0] = 0X0D;
 new_line[1] = 0X0A;
              /* set default file access: own read and write access, */
              /* read access for friends and others.                 */
 fmask = umask(S_IREAD | S_IWRITE | S_GREAD | S_PREAD);
               /* Create file "file1:symb" with default file access. */
               /* The file will be opened for read/write.            */
 if ((fn1=creat("file1:symb",fmask)) < 0) {
  /* on error: print the error and exit */
  perror("Creating file1:symb");
  exit(1);
 }
                          /* write leading text to the terminal      */
                          /* (new lines must be written explicitely) */
 write(fileno(stdout), out1, strlen(out1));
 write(fileno(stdout), &new_line[0], 2);
 write(fileno(stdout), out2, strlen(out2));
 write(fileno(stdout), &new_line[0], 2);
    /* While read doesn't return 0 or -1 read from the terminal into */
    /* the buffer, also take care of the actual buffer position      */
 bufpos = 0;
 while ((b_read = read(fileno(stdin), &buffer[bufpos], BUFLEN-bufpos-1))
       && (b_read != -1))
   bufpos = bufpos + b_read;
 if (!bufpos) {
  write(fileno(stdout), &new_line[0], 2);
  write(fileno(stdout), out_err, strlen(out_err));
  close(fn1);
  unlink("file1:symb");
  exit(1);
 }
              /* write the buffer with the terminal input to the file */
 b_written = write(fn1, &buffer[0], bufpos+1);
 fsync(fn1);       /* write the c-buffered data to the SINTRAN buffer */
 lseek(fn1, 0, 0); /* set the filepointer to the beginning of the file*/
 for (i=0; i<BUFLEN-1; i++) buffer[i] = ' ';          /* clear buffer */
                        /* read from the file into the internal buffer */
 b_read = read(fn1, &buffer[0], BUFLEN);
                                 /* write leading text to the terminal */
 write(fileno(stdout), &new_line[0], 2);
 write(fileno(stdout), out3, strlen(out3));
```

```
write(fileno(stdout), &new_line[0], 2);
    /* write the internal buffer to the terminal as long as it       */
    /* contains characters read from the file. If a CR occurs in the */
    /* buffer, add a line feed                                       */
 for (i=0; i<b_read; i++) {
  write(fileno(stdout), &buffer[i], 1);
  if (buffer[i] == 0X0D) write(fileno(stdout), &new_line[1], 1);
 }
   /* close and delete file file1:symb, associated with file no. fn1 */
 close(fn1);
 unlink("file1:symb");
}
```

BASIC-IO-3

```
/* A continuous file is created (Monitor Call CreateFile), opened     */
/* (with O_S3COM), written to (write) and truncated (to  bytes        */
/* needed).                                                           */
/* Its file status is inspected (stat, printf interesting things),    */
/* its access is changed (chmod to read-only), its last access time   */
/* is changed (utimes: last access to year ago, just to show how it   */
/* works).                                                            */
/* The last access time is compared to today's time (gettimeofday).   */
/* As the file seems to be too old now, its access is changed back    */
/* to read and write for everyone (chmod) and its name is changed     */
/* to OLD-<filename> (rename).                                        */
/* Occuring errors would be printed with printf and the help of err-  */
/* code (for the Monitor Call) or with the normal perror function.    */
/*                                                                    */
/* It looks funny if you inspect the file statistics under SINTRAN    */
/* after having changed the last access; the date of creation is      */
/* still uptodate, but that's the way it works!                       */
/*                                                                    */
#include <stdio.h>
#include <fcntl.h>
#include <types.h>
#include <stat.h>
#include <time.h>
#define BUFLEN 2048
                  /* Bits set to decode the SINTRAN access protection */
#define OWN_RD    0X0001
#define OWN_WR    0X0002
#define OWN_AP    0X0004
#define OWN_CM    0X0008
#define OWN_DR    0X0010
#define FRIEND_RD 0X0020
#define FRIEND_WR 0X0040
#define FRIEND_AP 0X0080
#define FRIEND_CM 0X0100
#define FRIEND_DR 0X0200
#define PUBLIC_RD 0X0400
#define PUBLIC_WR 0X0800
#define PUBLIC_AP 0X1000
#define PUBLIC_CM 0X2000
#define PUBLIC_DR 0X4000
```

```
#define YEAR_SECS (365*24*60*60)
extern short errcode();       /* the error function for Monitor calls */
static struct stat file_stat;
static struct timeval now, passed[2];

main()
{
 char *filename="CONTFILE:DATA";
 char *ptr1="1", *ptr2="2";
 char time_now[26], time_passed[26];
 char *tp;
 short mon_err;
 int fno, i;
                              /* create a continuous file with 5 pages */
 CreateFile(filename, 0, strlen(filename)-1, 0, 5);
 if (mon_err=errcode()) {
   printf("SINTRAN error %d in Monitor Call CreateFile\n", mon_err);
   exit(1);
 }
 if ((fno = open(filename, O_RDWR | O_S3CHAR |
                           O_S3NABBR | O_S3COM,0)) < 0) {
   perror("Open continuous file");
   exit(1);
 }
                                          /* write 2 pages - just junk */
 for (i=0; i<BUFLEN; i++) write(fno, ptr1, 1);
 for (i=0; i<BUFLEN; i++) write(fno, ptr2, 1);
 close(fno);
 if (truncate(filename, 2*BUFLEN)) {
   perror("Truncating continuous file");
   unlink(filename);
   exit();
 }
 stat(filename, &file_stat);
 printf("Interesting things from the file status of %s ",filename);
 printf("after truncating:\n");
 printf ("Acc. protection bits: ");
 outmode (file_stat.st_mode);
 printf ("SINTRAN access protection bits: \n");
 sin_outmode (file_stat.st_s3mode);
 printf ("Bytes in file        : %d\n", file_stat.st_size    );
 tp = ctime(&file_stat.st_atime);
 for (i=0; i<27; i++) time_now[i] = *tp++;      /* to save the string */
 printf ("Last read access     : %s", &time_now[0]);
 printf ("Last write access    : %s", ctime(&file_stat.st_mtime));
 printf ("Time of creation     : %s", ctime(&file_stat.st_ctime));
 printf ("Block size (2048)    : %d\n", file_stat.st_blksize);
 printf ("Allocated blocks     : %d\n", file_stat.st_blocks );
 gettimeofday(&now, NULL);
 passed[0].tv_usec = passed[1].tv_usec = now.tv_usec;
 passed[0].tv_sec  = passed[1].tv_sec  = now.tv_sec - YEAR_SECS;
```

```
 if (utimes(filename, &passed)) {
   perror("Utimes");
   exit(1);
 }
 if (chmod(filename, S_IREAD | S_GREAD | S_PREAD)) {
   perror("Changing mode to read only");
   unlink(filename);
   exit();
 }
 stat(filename, &file_stat);
 printf("Last-read after utimes; ");
 printf("SINTRAN access after chmod to read-only:\n");
 tp = ctime(&file_stat.st_atime);
 for (i=0; i<27; i++) time_passed[i] = *tp++;   /* to save the string */
 printf ("Last read access    : %s", &time_passed[0]);
 sin_outmode (file_stat.st_s3mode);
 if ((atoi(&time_now[20]) - atoi(&time_passed[20])) > 0) {
                            /* last read access more than 1 year ago */
   chmod("CONTFILE:DATA", S_IREAD | S_IWRITE | S_GREAD | S_GWRITE |
                            S_PREAD | S_PWRITE);
   if (rename(filename, "OLD-CONTFILE:DATA")) {
     perror("Rename to OLD-...");
     exit(1);
   }
   printf("The file was renamed to OLD-CONTFILE:DATA\n");
 }
}

static outmode (m) int m;
{
 printf("PUBLIC ");
 if (m & S_PREAD ) printf ("READ");
 if (m & S_PWRITE) printf ("/WRITE");
 printf(": FRIEND ");
 if (m & S_GREAD ) printf ("READ");
 if (m & S_GWRITE) printf ("/WRITE");
 printf(": OWN ");
 if (m & S_IREAD ) printf ("READ");
 if (m & S_IWRITE) printf ("/WRITE");
 printf ("\n");
}

static sin_outmode (sm) int sm;
{
 printf("PUBLIC ACCESS: ");
 if (sm & PUBLIC_RD) printf ("READ ");
 if (sm & PUBLIC_WR) printf ("WRITE ");
 if (sm & PUBLIC_AP) printf ("APPEND ");
 if (sm & PUBLIC_CM) printf ("COMMON ");
 if (sm & PUBLIC_DR) printf ("DIRECTORY");
```

```
printf("\nFRIEND ACCESS: ");
if (sm & FRIEND_RD) printf ("READ ");
if (sm & FRIEND_WR) printf ("WRITE ");
if (sm & FRIEND_AP) printf ("APPEND ");
if (sm & FRIEND_CM) printf ("COMMON ");
if (sm & FRIEND_DR) printf ("DIRECTORY");
printf("\nOWN ACCESS:    ");
if (sm & OWN_RD) printf ("READ ");
if (sm & OWN_WR) printf ("WRITE ");
if (sm & OWN_AP) printf ("APPEND ");
if (sm & OWN_CM) printf ("COMMON ");
if (sm & OWN_DR) printf ("DIRECTORY");
printf ("\n");
}
```

Standard Functions

Formatted I/O

The functions of the formatted I/O system work via an own buffer system. When reading with formatted input functions and the buffer of the formatted I/O system is empty, the basic input function *read* will be called internally. The equivalent applies for output; when the buffer of the formatted I/O system is full, the basic output function *write* will be called internally. The chart in Appendix C on page 17-81 shows the relationship between functions of the basic I/O system and functions of the formatted I/O system.

In the formatted I/O system, there are three types of buffering:

unbuffered

- When an output file is unbuffered, the data are immediately transferred to the operating system.

block-buffered

- When an output file is block-buffered, 2048 characters are saved up and written as a block.

line-buffered

- When an output file is line-buffered, characters are saved up until a carriage return (\n) is encountered or input is read from the standard input file *stdin*.

By default, the functions described in this section treat the files as block-buffered, with a buffer size of 2048 bytes. The buffer is allocated upon the first read or write on a file. The functions *setbuf* and *setbuffer* (see page 13-44) allow you to change the buffer handling.

FILE structure

Before you can access a file you have to connect the physical file to your program. This can be done by calling the function *fopen*, which dynamically returns a pointer to an internal structure FILE. This structure, declared in the header file *stdio.h*, contains file information, such as the location of the buffer, the current position of the buffer pointer, the access mode, etc. The file pointer returned by *fopen* is your connection to the file and must be used when accessing or manipulating the file.

stream	In the following, the physical file together with its FILE information is referred to as <u>stream</u>.
header file *stdio.h*	Any program using buffered I/O functions must include the header file *stdio.h*:

#include <stdio.h>

This file contains the declarations of the functions described in this section, the type definition of above described structure FILE and also the following declarations:

standard open files

- Four pointers to standard open files:

file pointer	file name
stdin	standard input file
stdout	standard output file
stderr	standard error file
stdtmp	standard temporary file

NULL

- The constant NULL (0):
  A file pointer with the value NULL designates no file at all.

EOF

- The integer constant EOF (-1):
  The value EOF is returned upon end of file or to indicate that a FILE pointer has not been intialised with *fopen*, input (output) has been attempted on an output (input) file, or a file pointer designates unintelligible FILE data.

Above constants and the following "functions" are implemented as macros: *fgetc*, *getchar*, *fputc*, *putchar*, *feof*, *ferror*, *clearerr*, *fisbinary* and *Fileno*. Redeclaration of these names causes an error.

All functions of the standard buffered I/O package may be freely intermixed.

FUNCTION open a file: *fopen*, *freopen*, *fdopen*

HEADER FILE #include <stdio.h>

DECLARATION

- ```
  FILE *fopen (fname, mode);
  char *fname, *mode;
  ```
- ```
  FILE *freopen (fname, mode, fp);
  char *fname, *mode;
  FILE *fp;
  ```
- ```
  FILE *fdopen (ldn, mode);
  int ldn;
  char *mode;
  ```

DESCRIPTION

fopen

Via a call to the system function **open** the function *fopen* opens the file named by *file*, allocates a FILE structure and returns a file pointer *fp* to it. The default file type is :SYMB. The file pointer returned is used as a file reference in buffered access functions like *getc*, *fread*, *setbuf*, etc.

freopen

The function *freopen* works as *fopen* with the difference that the file pointer is not supplied by the runtime system, but by the user as third parameter *fp*. Before the function is executed the previous file associated with this pointer is closed. A typical application of this function is to redirect the standard input or output files. For this purpose you just specify *stdin* or *stdout* as file pointer.

fdopen

The function *fdopen* works as *fopen*, but instead of a file name you specify the file number obtained from *open*, *creat*, *dup* or *dup2* (see pages 13-13, 13-13 and 13-19). If you want to open a file, which is already open on system level, you have to specify the same *mode*.
  ```

mode

The parameter *mode* specifies the access rights and whether your file is to be handled as a text file or as a binary file (see below):

"r" open for reading: positions the file pointer at the beginning of the file.

"w" open for writing: creates a file if not yet created or truncates an existing file to zero length.

"a" append: opens for writing at end of file, or, if not yet created, creates for writing.

"r+" open for update (reading and writing).

"w+" truncate or create for update.

"a+" open or create for update at end of file.

name conflicts

When using the upper case letters "R", "W" and "A" instead of the lower case letters, an exact match of the file name is required. In this way the user can avoid name conflicts due to the abbreviation facility in SINTRAN.

text files

Furthermore, the SINTRAN file system differs from C and UNIX in the handling of text files. It uses parity bits in characters where C uses none. This means, that text files have to be converted by the library when reading or writing. This conversion is not needed for binary files. By default files are handled as text files. To

binary files

indicate that a file is to be considered as a binary file you add an "b" to the mode parameter, e.g. "rb", "W+b" or "a+b".

file as segment

To indicate that a file is to be opened as segment a *s* is added to the mode parameter, for example "ws". See also *O_S3SEG* on page 13-13.

RETURN VALUE

fopen, *freopen* and *fdopen* return a pointer to the FILE structure associated with the file opened. The pointer value NULL indicates an error.

NOTES

The use of "b", "s","W", "A" and "R" in the *mode* parameter is a special extension for the SINTRAN operating system and as such non-portable.

Examples: *fopen* on page 13-48, *freopen* on page 13-113.

FUNCTION buffer handling: *setbuf*, *setbuffer*

HEADER FILE #include <stdio.h>

DECLARATION

```
● void setbuf (fp, buf);

  FILE *fp;
  char *buf;

● void setbuffer (fp, buf, size)
  FILE *fp;
  char *buf;
  int size;
```

DESCRIPTION

setbuf

The function *setbuf* allocates a character array of 2048 bytes starting at *buf* instead of an automatically allocated buffer. If *buf* equals NULL the input or output will be completely unbuffered.

setbuffer

setbuffer works the same as *setbuf*, with the extension, that you can determine the size of the buffer yourself.

NOTES

These functions should only be called after a file has been opened and before the first read or write access. The four standard files have not to be opened as they are open by default.

You will find an example of *setbuf* on page 13-113.

FUNCTION — reposition the file pointer: *fseek*, *rewind*, *ftell*

HEADER FILE — **#include** <stdio.h>

DECLARATION

- **int** fseek (fp, offset,position);
  FILE *fp;
  long offset;
  int position;

- **void** rewind (fp);
  FILE *fp;

- **long** ftell (fp);
  FILE *fp;

DESCRIPTION

fseek — *fseek* sets the position of the buffer pointer *offset* bytes from the beginning (position), from the current position (*position=1*) or from the end of the file (*position=2*). For *offset* negative values may be specified as well as positive.

rewind — *rewind(fp)* is equivalent to *fseek(fp,0L,0)*, except that no value is returned. It sets the file pointer to the beginning of the file.

ftell — *ftell* returns the offset of the current buffer pointer relative to the beginning of the file associated with *fp*. The offset is measured in bytes.

RETURN VALUE — On error *fseek* returns the value -1, otherwise zero.

NOTES — An error can occur, when the file has not been opened using *fopen*. Furthermore, *fseek* must not be used on files which are opened for sequential access only (e.g. terminals).

You will find an example of *fseek* on page 13-48.

FUNCTION

flush buffered data and close a file: *fclose*
flush buffered data: *fflush*

HEADER FILE

#include <stdio.h>

DECLARATION

- ```
  int fclose (fp);
  FILE *fp;
  ```
- ```
  int fflush (fp);
  FILE *fp;
  ```

DESCRIPTION

fclose

The function *fclose* frees the buffer and closes the file associated with *fp*. If it is an output file all buffered data will be written to the file before closing it.

Upon calling *exit* (see page 13-112) *fclose* is performed automatically for all open files.

fflush

The function *fflush* causes any buffered data belonging to the specified file pointer *fp* to be output, but contrary to *fclose*, *fflush* leaves the file open. As such, *fflush* is only meaningful on output files; it has no effect on input files. Typically, *fflush* is used to output intermediate messages to the terminal while the program is running and the line buffer is not yet full. If the output file is on a permanent storage device (disk) and to ensure that all data is on the file at that moment, it additionally may be necessary to call the basic function *fsync*, which flushes the buffer of the basic I/O system.

RETURN VALUE

Both functions return 0 for success; on error -1 is returned.

You will find an example of *fclose* on page 13-64.

FUNCTION

file status inquiries:
ferror, feof, clearerr, fileno, fisbinary

HEADER FILE

#include <stdio.h>

DECLARATION

- **int** ferror (fp);
  FILE *fp;
- **int** feof (fp);
  FILE *fp;
- **void** clearerr (fp);
  FILE *fp;
- **int** fileno (fp);
  FILE *fp;
- **int** fisbinary (fp);
  FILE *fp;

DESCRIPTION

ferror

ferror returns non-zero when an error has occurred reading or writing the specified file, otherwise zero. Unless cleared by *clearerr*, the error indication lasts until the file is closed.

feof

feof returns non-zero when the end of the specified file is reached, otherwise zero.

clearerr

clearerr resets the error indicators of *ferror* and *feof* for the specified file to zero.

fileno

fileno returns the UNIX file number of the specified file which except for very special exceptions is identical to the SINTRAN file number.

fisbinary

fisbinary returns non-zero if the file associated with *fp* is a binary file, otherwise 0. A file is regarded as binary if it has been opened as binary.

NOTES

ferror, feof, clearerr and *fileno* are implemented as macros. Remember, that you must not redeclare macros in your program.

example

```
#include <stdio.h>

main()
{
  FILE *in;
  char str[255];
  int max=255;
  char file_name[32];

  printf("Enter filename : ");
  gets(file_name);
  printf("\n\n");

  if (!(in = fopen(file_name, "r"))) {
    printf("Error in fopen! Program halts!\n");
  } /* if */
  else {
    printf("File %s - SINTRAN file-number %d - is open for read.\n",
            file_name, fileno(in));
    if (fisbinary(in))
      printf("%s is a binary file", file_name);
    else {
      printf("Reading will start at position 4.\n");
      fseek(in, 4, 0);
      while (!feof(in)) {
        fgets(str, max, in);
        if (!ferror(in))
          if (!feof(in)) fputs(str, stdout);
      } /* while */
      printf("End of file at position %ld.\n", ftell(in));
    } /* else (fisbinary) */
  } /* else */
}
```

FUNCTION	get a character or word: *getc*, *fgetc*, *getchar*, *getw*
HEADER FILE	#include <stdio.h>

DECLARATION

- ```
  int getc (fp);
  FILE *fp;
  ```
- ```
  int fgetc (fp);
  FILE *fp;
  ```
- ```
  int getchar ();
  ```
- ```
  int getw (fp);
  FILE *fp;
  ```

DESCRIPTION

getc — *getc* returns the next character from the specified input stream pointed to by *fp*.

fgetc — *fgetc* performs the same function as *getc*. As it is a macro you cannot define a pointer to it. If you need to use a pointer you should use *getc*.

getchar — *getchar* is identical to *getc(stdin)*. It is implemented as a macro.

getw — *getw* returns the next word of the specified file, which is 4 bytes long on a ND-500. As EOF is a valid integer value, you should use *feof* and *ferror* to check on error and end of file. The function *getw* assumes no special alignment in the file, i.e. it can be intermixed with *getc*, *getchar*, *fgets* without restrictions.

RETURN VALUE — All four functions return EOF upon end of file or error.

Examples: *getc* on page 13-49, *getchar* on page 13-64.

FUNCTION

push a character back into input stream: *ungetc*

HEADER FILE

#include <stdio.h>

DECLARATION

- **int** ungetc (c,fp);
  char c;
  FILE *fp;

DESCRIPTION

Ungetc replaces the last character read from the input buffer associated with *fp* by *c* and sets the buffer pointer one byte backwards. The character *c* will be returned by the next *getc* call on that file.

RETURN VALUE

When the call fails, *ungetc* returns EOF. Otherwise, *c* is returned.

NOTES

This function works only, if something has been read before and the stream is actually buffered. The only exception is the standard input file *stdin*, which allows you to insert exactly one character into the buffer without a previous input statement.

The functions *fseek* and *rewind* (see page 13-45) erase all memory of characters replaced.

For an example of ungetc see page 13-50.

FUNCTION

write a character or word: *putc*, *fputc*, *putchar*, *putw*

HEADER FILE

#include <stdio.h>

DECLARATION

- ```
  int putc (c,fp);
  char c;
  FILE *fp;
  ```

- ```
  int fputc (c,fp);
  char c;
  FILE *fp;
  ```

- ```
  int putchar (c);
  char c;
  ```

- ```
  int putw (w,fp);
  int w;
  FILE *fp;
  ```

DESCRIPTION

putc

putc appends the character *c* to the file associated with *fp*. It returns the character written.

fputc

fputc performs the same function as *putc*, but is implemented as a macro. Therefore, you cannot define a pointer to it.

putchar

The macro *putchar(c)* is equivalent to *putc(c, stdout)*.

putw

putw appends the word *w* (4 bytes on a ND-500) to the file associated with *fp*. It neither assumes nor causes special alignment in the file, i.e. it may be intermixed with *putc*, *putchar*, *fputc* without restrictions. A successful call returns the word written. As the error return value EOF is a valid integer, *ferror* should be used to detect errors.

RETURN VALUE

After successful execution these functions each return the value written. On error EOF is returned.

NOTES

The use of *putw* makes a program less portable. Files written using *putw* are machine-dependent and cannot be read using *getw* on a processor with a different word length or byte ordering.

You will find an example of *putc* on page 13-64.

FUNCTION get a string: *gets*, *fgets*

HEADER FILE #include <stdio.h>

DECLARATION

```
• char *gets (s);
  char *s;

• char *fgets (s, n, fp);
  char *s;
  int n;
  FILE *fp;
```

DESCRIPTION

gets

gets reads a string from the standard input file into the array pointed to by *s* until a carriage return is encountered. The carriage return (\n) is not transferred into the array, but replaced by a null character (\0).

fgets

fgets reads at most *n-1* characters from the file associated with *fp* into the array pointed to by *s*.

The transfer stops, when EOF or a carriage return (\n) is encountered. Contrary to *gets* a carriage return is moved into the array. A null character is written immediately after the last character read into the array.

RETURN VALUE

Both functions return the constant pointer value NULL upon end of file or error. Otherwise, the unmodified *s* will be returned.

Examples: *gets* on page 13-48, *fgets* on page 13-48.

FUNCTION write a string: *puts*, *fputs*

HEADER FILE #include <stdio.h>

DECLARATION

```
● int puts (s);
  char *s;

● int *fputs (s, fp);
  char *s;
  FILE *fp;
```

DESCRIPTION

puts

puts copies the string *s* to the standard output file *stdout*. Instead of the null character at the end of the string a carriage return (\n) is transferred.

fputs

fputs copies the string *s* to the file associated with *fp* without the terminating null character. Contrary to *puts* the null character is not replaced either.

RETURN VALUE

Both functions return EOF on error. This will happen, if you try to write to a file that has not been opened for writing.

NOTES

Neither function copies the terminating null character. Note that *puts* appends a carriage return character, while *fputs* does not.

You will find an example of *fputs* on page 13-48.

FUNCTION

array input and output: *fread*, *fwrite*

HEADER FILE

#include <stdio.h>

DECLARATION

- ```
  int fread (ptr, size, nitems, fp);
  char *ptr;
  int size, nitems;
  FILE *fp;
  ```

- ```
  int fwrite (ptr, size, nitems, fp)
  char *ptr;
  int size, nitems;
  FILE *fp;
  ```

DESCRIPTION

fread

The function *fread* reads *nitems* items from the input stream associated with *fp* into the program-internal array pointed to by *ptr*. An item is a sequence of *size* bytes, which may possibly be terminated by a null byte.

fwrite

The function *fwrite* writes at most *nitems* items from the array pointed to by *ptr* to the output stream associated with *fp*. A common way of specifying the *size* of an item is the construct **sizeof** *(*ptr)*, where **sizeof** gives the length of the item pointed to by *ptr*. if *ptr* points to a data type other than **char** a cast construct for explicit type conversion should be used (see page 3-16).

RETURN VALUE

Fread and *fwrite* return the number of items read or written, which may be less than *nitems* if an I/O error or an end-of-file is encountered. If *size* or *nitems* is zero, zero is returned and the state of the stream remains unchanged.

You will find an example of *fwrite* on page 13-66.

FUNCTION

formatted input conversion: *scanf*, *fscanf*, *sscanf*

HEADER FILE

#include <stdio.h>

DECLARATION

- **int** scanf (format [, pointer] ...);
  char *format;

- **int** fscanf (fp, format[, pointer] ...);
  FILE *fp;
  char *format;

- **int** sscanf (s, format[, pointer] ...);
  char *s, *format;

DESCRIPTION

scanf reads from the standard input stream *stdin*. *fscanf* reads from the specified input stream *fp*. *sscanf* reads from the character string *s*.

Each function reads characters, interprets them according to the control string specified in *format*, and stores the results in the remaining parameters. The control string is described below; the other parameters, each of which must be a pointer, indicate where the converted input should be stored.

control string

The control string, which is enclosed by quotes, may contain:

- <u>white space characters</u> like blanks, tabs or carriage returns, which match optional white space in the input and are ignored,

- <u>ordinary characters</u> (not %), which must match the next input character, and

- <u>conversion specifications</u>, consisting of
  1. the character %
  2. an optional assignment suppression character *
  3. an optional number specifying a maximum field width
  4. a conversion character (see below)

  A conversion specification directs the conversion of the next input field. The result is placed in the variable pointed to by the corresponding parameter. If assignment suppression is indicated by the * character, the input field is skipped and no assignment is made. For a suppressed field no parameter should be given.

> **Note**
> The parameters after the control string <u>must</u> be pointers. Otherwise the result is undefined; you will not get a proper error message.

input field
: An input field is defined as a string of non-white space characters. It extends either to the next white space character or until the specified field width is exhausted.

conversion character
: The conversion character indicates the interpretation of the input field. The following conversion characters are legal:

d — A decimal integer is expected; the corresponding parameter should be an integer pointer.

u — An unsigned decimal integer is expected; the corresponding parameter should be an unsigned integer pointer.

o — An octal integer is expected; the corresponding parameter should be an integer pointer.

x — A hexadecimal integer is expected; the corresponding parameter should be an integer pointer.

e,f,g — A floating point number is expected; the corresponding parameter must be a pointer to float. The input format for floating point numbers equals the syntax of an optionally signed floating constant (see page 2-8).

c — A single character is expected; the corresponding parameter should be a character pointer. In this case the normal skip over white space characters is suppressed; to read the next non-white space character, use the conversion specification %1s.

s — A character string is expected; the corresponding parameter should be a character pointer pointing to an array of characters large enough to accept the string and a terminating \0 which will be added.

[..] — The square brackets indicate a string not to be delimited by white characters. The left bracket is followed by a set of characters, which we call the scanset, and a right bracket. The corresponding parameter must point to a character array large enough to hold the data field plus the terminating \0, which will be added automatically.

If the first character in the scanset is not a circumflex (^), the input field is all characters until the first character which is not part of the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character only being preceded by a possibly needed circumflex.

If the first character in the scanset is a circumflex (^), the input field is defined as all characters until the first character which is found in the scanset. A range of characters may be represented by the construct *first - last*, where the first character must have a lower ASCII-value than the last.

The conversion characters *d*, *u*, *o* and *x* may be preceded by l (long) or h (short) to indicate that the parameter is **long** or **short** rather than **int**.

To indicate that a parameter of type **double** is expected, *e*, *f* or *g* may be preceded by the letter l.

If the character after the percentage sign is not a conversion character, this specified character is expected in the input, but no assignment is made.

The conversion terminates at the end of input, at the end of the control string, or when an input character conflicts with the control string. In the latter case the conflicting character is left unread in the input stream.

RETURN VALUE

For all three functions, the return value is the number of successfully matched and assigned input items; the number is zero if an early conflict between an input character and the control string occurs. If the input ends before the first conflict or conversion EOF is returned.

NOTES

Trailing white space is left unread unless matched in the control string.

examples

- ```
  int i;
  float x;
  char name [50];

  scanf ("%d %f %s", &i, &x, name);
  ```

  The input line

  25 54.32E-1 thompson

  will cause the following assignments:

  ```
  i    = 25
  x    = 5.432
  name = thompson\0
  ```

- ```
  int i;
  double x;
  char name [50];

  scanf ("%2d %lf *d %[0-9]", &i, &x, name);
  ```

  The input line

  56789 0123 56a72

  will cause the following assignments:

  ```
  i    = 56
  x    = 789.0
  0123 will be skipped
  name = 56\0
  ```

  A following call to *getchar* would return the character *a*.

FUNCTION — print formatted output: *printf*, *fprintf*, *sprintf*

HEADER FILE — **#include** <stdio.h>

DECLARATION

- **int** printf (format [, parm] ...);
  char *format;

- **int** fprintf (fp, format[, parm] ...);
  FILE *fp;
  char *format;

- **int** sprintf (s, format[, parm] ...);
  char *s, *format;

DESCRIPTION

printf — *printf* places its output on the standard output stream *stdout*.

fprintf — *fprintf* places its output on the file associated with the file pointer *fp*.

sprintf — *sprintf* places its output in the string *s*, adding the terminating character \0.

control string — All of these functions convert, format and print their parameters according to the control string specified in *format*. This control string, which is enclosed by quotes, contains two types of objects:

- strings, which are simply copied to the output stream, and

- conversion specifications, each of which causes conversion and printing of the next successive parameter.

The number and types of conversion specifications and remaining parameters should be the same. If there are not enough remaining parameters or if they point to the wrong type, the result is undefined.

conversion specification — Each conversion specification is introduced by the character %. The percentage sign is followed by:

flag — 1. one or more optional flags, which modify the meaning of the conversion specification (see page 13-62).

field width — 2. an optional decimal digit string specifying the minimum field width. If the converted value has fewer characters than the field width, it will be blank-padded on the left (or on the right, if left adjusted); if the field width begins with a zero, zero-padding will be done instead. In no case a non-existent or too small field width will cause truncation of a field; the field is expanded. Instead of the digit string you may also specify an asterisk (see description under precision below).

precision — 3. an optional digit string specifying a precision that, depending on the conversion character, gives:

- for the formats *d*, *o*, *u*, *x* or *X*:
  the minimum number of digits to appear

- for the formats *e* and *f*:
  the number of digits to appear after the decimal point

- for the *g* format :
  the maximum number of significant digits

- for the *s* format (strings):
  the maximum number of characters to be printed

The precision is written as a period (.) followed by an optional decimal digit string. If the digit string is omitted the precision is treated as zero.

The field width and precision may also be indicated by an asterisk (*). In this case the values must be given before the corresponding parameter. This may be useful if you want to use the same printing command, but with different precisions and/or field width; the values may be specified as variables.

example

```
printf ("%*.*d\n", 10, 4, 1.2345);

                /* field width = 10 */
                /* precision   = 4  */
```

l (long)
h (short) — 4. an optional l or h specifying that a following conversion character *d*, *o*, *u*, *x* or *X* applies to a **long** respectively a **short** integer parameter.

conversion character — 5. a conversion character, which indicates the type of conversion to be applied.

conversion characters — The conversion characters and their meanings are:

d,o,u,x,X — The integer parameter is converted to signed decimal, octal, unsigned or hexadecimal notation respectively. The lower case conversion character *x* for hexadecimal notation specifies that the lower case letters a..f have to be used, whereas *X* specifies that the upper case letters A..F have to be used. The default precision is 1.

f — A **float** or **double** parameter is converted to decimal notation of the form [-]mmm.nnn, where the number of digits after the decimal point is specified with the precision. Note that in this case the precision does not determine the number of significant digits. As default 6 digits are output after the decimal point; if the precision equals zero, no decimal point appears.

e,E — A **float** or **double** parameter is converted to exponential notation of the form [-]m.nnne±dd, where there is one digit before the decimal point and the number of digits after the point (nnn) is equal to the precision specified. As default 6 digits are output after the decimal point. The exponent consists of at least two digits. Whether you specify a capital *E* or *e* just determines which of these letters will introduce the exponent.

g,G — A **float** or **double** parameter is converted to *f*, *e* or *E* format, with the precision specifying the number of significant digits. The format used depends on the value of the parameter; format *e* (or *E*, if *G* is specified) will only be used, if the exponent is less than -4 or greater than or equal to the precision. Non-significant zeroes are not printed. A decimal point appears only, if it is followed by a digit.

c — The parameter is taken to be a single character.

s — The parameter is taken to be a string. Characters from the string are printed until a null character (\0) is encountered or the number of characters specified with the precision is reached. If no precision is explicitly specified all characters up to the first null character are printed. If the pointer parameter equals NULL, the result is undefined.

A character after the percentage sign which is not a conversion character, is printed. Thus a percentage sign may be printed by %%.

flags The flag characters and their meanings are:

- \- The result of the conversion will be left-justified within the field.

- \+ The result of a signed conversion will always begin with a sign (+ or -).

- *blank* If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies, that if the blank and + flag both appear, the blank flag will be ignored.

- *0* The output field will be filled up with zeroes instead of blanks.

- \# This flag specifies that the value is to be converted to an "alternate form":

  - For the formats *d*, *u*, *c* and *s* this flag has no effect.
  - For the format *o*, it increases the precision to force the first digit of the result to be a zero.
  - For the format *x* or *X*, a non-zero result will be preceded by 0x resp. 0X.
  - For the formats *e*, *E*, *f*, *g* and *G*, the result will always contain a decimal point, even if no digits follow the point.
  - For the formats *g* and *G*, non-significant zeroes will not be removed from the result.

If more than one flag is needed, they have to be specified in the order described above.

RETURN VALUE After successful execution the number of transmitted characters will be returned (not including the \0 in the case of sprintf) or a negative value if an output error occurred.

examples

- The first example shows which effect different precisions have as related to strings. For clarity's sake, the field boundaries are marked by colons. The field to be printed "hello, world" consists of 12 characters.

```
:%10s:      :hello, world:
:%-10s:     :hello, world:
:%20s:      :        hello, world:
:%-20s:     :hello, world        :
:%20.10:    :          hello, wor:
:%-20.10s:  :hello, wor          :
:%.10s:     :hello, wor:
```

- The following program can be used to print a date and time in the form "Sunday, July 3, 10:02":

```
main()
{
  char weekday[20], month[20];
  int day, hour, min;

  printf("weekday? ");
  scanf("%s", weekday);

  printf("month? ");
  scanf("%s", month);

  printf("day? ");
  scanf("%d", &day);

  printf("hour? ");
  scanf("%d", &hour);

  printf("min? ");
  scanf("%d", &min);

  printf("%s, %s %d, %.2d:%.2d",weekday,month,day,
         hour,min);
}
```

Do not forget to define the starting address of a character array, which you use as a pointer; otherwise, the result will be undefined. You will not get an error message! In this example the starting addresses of *weekday* and *month* are defined by giving the size within the declaration.

- The following example shows the effect of different precisions and conversion characters on numbers:

```
main()
{
  double fl;
  int i;

  fl=12.2345;  i=78;
  printf(":%+3.2f:     :%03d:\n",fl,i);

  fl=74653.2;  i=7898;
  printf(":%+4.2f:  :%3d:\n",fl,i);
}
```

Output:

```
:+12.23:     :078:
:+74653.20:  :7898:
```

Examples: *sprintf* on page 15-61, *fprintf* on page 13-113.

```
/* This program gets characters from the terminal (up to a pro-     */
/* grammed end) and writes them into a file which isn't existing    */
/* yet. The characters put into the file are counted.               */
/* Then the file is read characterwise and it's contents are given  */
/* as output on the terminal again.                                 */
/*                                                                  */
/* NOTE: although the end of getchar is programmed to be '$$', a    */
/*       carriage return has to follow, because input from the      */
/*       terminal is always line buffered.                          */
/*                                                                  */
/* Calls of: fopen, getchar, getc, putchar, putc, printf, fclose,   */
/*           unlink, mktemp.                                        */
#include <stdio.h>
extern char *mktemp();
main()
{
 FILE *iop;
 char *file_name = "file000001:symb";
 char *out1 = "Type what you would like to be written to file1!";
 char *out2 = "(End your input with a double $-sign: $$+<CR>)";
 char *out3 = "Your input, read from file1:";
 char *out_err      = "That was no input!";
 int i, j, c;
          /* Test if "file000001:symb" already exists, if it does, */
          /* take another name.                                    */
 if ((iop = fopen(file_name,"R"))) {
  printf("File \"%s\" already exists,\n", file_name);
  printf("will open another file with a unique file name!\n");
  fclose(iop);
  file_name = mktemp("fileXXXXXX:symb");
  if (strcmp(file_name,"") == 0) {
   printf("Sorry, no unique file name can be built\n");
   exit(1);
  }
  printf("Your file will have the name \"%s\"\n", file_name);
 }
 /* Open text file file_name for write. */
 if (!(iop = fopen(file_name,"w"))) {
  printf("The file \"%s\" cannot be opened\n", file_name);
  exit(1);
 }
                         /* The file will be opened for read/write. */
                         /* Write leading text to the terminal      */
 printf("%s\n",out1);
 printf("%s\n",out2);
      /* Loop: read characters from the terminal and write them to */
      /* the file as long as no 2 subsequent $ occur:              */
 j = 0;
 read_write_loop:
 while ((c = getchar()) != '$') {
 if (putc(c, iop) == EOF) {
  printf("ERROR with putc\n");
  break;
  }
```

```
 if (c == '\n') j++;   /* In case of text files: one 'get..' reads */
                       /* 0X0D0A, and one 'put..' writes it.       */
 j++;
}
if ((i = getchar()) != '$') {
 putc(c, iop); j++;
 if (c == '\n') j++;
 putc(i, iop); j++;
 if (i == '\n') j++;
 goto read_write_loop;
}
if (!j) {
 printf("\n%s\n", out_err);
 fclose(iop);
 unlink(file_name);
 exit(1);
}
fclose(iop);
fopen(file_name, "R");
printf("(%d characters were written to \"%s\")\n", j, file_name);
printf("\n%s\n", out3);
while ((c = getc(iop)) != EOF) putchar(c);
fclose(iop);
printf("\n\"%s\" is not deleted yet!!!\n", file_name);
}
```

```
/* This program reads an address from the terminal and writes it    */
/* (together with the leading text) to a file which isn't existing   */
/* yet.                                                              */
/*                                                                   */
/* Calls of: fopen, fclose, fread, fwrite, mktemp, printf,           */
/*           strcmp, strlen, exit                                    */
#include <stdio.h>
extern char *mktemp();

FILE *iop;
main()
{
 struct address {
                 char first_name[20];
                 char last_name[25];
                 char street_no[6];
                 char street[25];
                 char city[25];
                 char city_code[8];
                 char country[25];
                };
 struct address addr = {"                    ",
                        "                         ",
                        "      ", "                         ",
                        "                         ", "        ",
                        "                         " };
```

```
char *file_name = "file000001:symb";
char *out1 = "Type name and address as leaded by the text\n";
char *out2 = "(End the single items with <CR>)\n";
char *out_fn = "First name   : ";
char *out_ln = "Last  name   : ";
char *out_sno= "Street number: ";
char *out_st = "Street name  : ";
char *out_c  = "City         : ";
char *out_cc = "City code    : ";
char *out_cn = "Country      : ";
int i, j, c;
          /* Test if "file000001:symb" already exists, if it does, */
          /* take another name.                                    */
if ((iop = fopen(file_name,"R"))) {
 printf("File \"%s\" already exists,\n", file_name);
 printf("will open another file with a unique file name!\n");
 fclose(iop);
 file_name = mktemp("fileXXXXXX:symb");
 if (strcmp(file_name,"") == 0) {
  printf("Sorry, no unique file name can be built\n");
  exit(1);
 }
 printf("Your file will have the name \"%s\"\n",file_name);
}
                              /* Open text file file_name for write. */
if (!(iop = fopen(file_name,"w"))) {
 printf("The file \"%s\" cannot be opened\n", file_name);
 exit(1);
}
                               /* Write leading text to the terminal */
fwrite(out1, 1, strlen(out1), stdout);
fwrite(out2, 1, strlen(out2), stdout);
     /* For each item call a routine that reads the terminal input */
     /* and writes it to the specified file.                       */
do_r_and_w(out_fn, &addr.first_name[0], 20);
do_r_and_w(out_ln, &addr.last_name[0], 25);
do_r_and_w(out_sno, &addr.street_no[0], 6);
do_r_and_w(out_st, &addr.street[0], 25);
do_r_and_w(out_c, &addr.city[0], 25);
do_r_and_w(out_cc, &addr.city_code[0], 8);
do_r_and_w(out_cn, &addr.country[0], 25);
fclose(iop);
printf("In file \"%s\" you may look up the result.\n", file_name);
}

     /* Routine which reads terminal input and writes it to a file */
     /* (after having written a leading text to the file).         */
void do_r_and_w(cptr1, cptr2, len)
char *cptr1, *cptr2;
int len;
```

```
{
 char *junk, jc;
 char *hptr;
 int i;
 junk = &jc;

 fwrite(cptr1, 1, 15, stdout);
 fwrite(cptr1, 1, 15, iop);
 hptr = cptr2;
 i = 0;
 do {
  fread(cptr2, 1, 1, stdin);
  i++;
 } while ((*cptr2++ != '\n') && (i<len));
     /* If the input exceeds a certain length, just ignore the rest */
 if ((i == len)  && (--*cptr2 != '\n'))
  do fread(junk, 1, 1, stdin); while (*junk != '\n');
 fwrite(hptr, 1, i, iop);
 if ((i == len)  && (*cptr2 != '\n'))
  fwrite("\n",1,1,iop);    /* To have a "new line" behind each item */
}
```

— Storage Allocation

The storage allocation functions allow you to dynamically allocate and deallocate heap space.

FUNCTION *malloc, free, realloc, calloc*

DECLARATION

- ```
  char *malloc (size);
  unsigned size;
  ```
- ```
  void free (ptr);
  char *ptr;
  ```
- ```
  char *realloc (ptr, size);
  char *ptr;
  unsigned size;
  ```
- ```
  char *calloc (nelem, elsize);
  unsigned nelem, elsize;
  ```

DESCRIPTION

malloc

malloc allocates a block of at least *size* bytes beginning on a word boundary and returns a pointer to this block. Rather than allocating from a compiled-in fixed-sized array, *malloc* will request space from the operating system as needed. It maintains lists of free blocks according to size and tries to get more memory from the system, when there is not enough contiguous space available. A typical example of how to use *malloc* is given on page 5-7.

free

The function *free* deallocates the memory space pointed to by *ptr*. Generally, it is used to free the space previously allocated by *malloc* using the pointer returned by *malloc* as parameter. The space freed is made available for further allocation, but its contents is not deleted.

realloc

realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents remains unchanged up to the smaller of the new and old sizes.

calloc

calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialised to zero.

RETURN VALUE

Each of the allocation functions returns a pointer to the space allocated, and which is suitably aligned for storage of any type of object, if there is memory available. By default allocation is done by dynamically expanding, i.e. the first segment will be allocated with 128 Kbytes (default value). If this space is exhausted a second segment would be allocated with 256 Kbytes and so on.

If there is no more space available you will get an overflow and no return value for the functions. If you want to have a fixed heap size and NULL pointers as return values if the space is exhausted you have to define this fixed size within the linkage loader with the following commands:

```
NLL: DEFINE-ENTRY fixed_heap <no. of possibly alloc. segments> d↵
NLL: DATA-REFERENCE fixed_heap rts_fixed_heap d↵
NLL: DEFINE-ENTRY heap_size <size in bytes> d↵
NLL: DATA-REFERENCE heap_size rts_heap_size d↵
```

These commands have to be given after the loading of the libraries. "rts_fixed_heap" and "rts_heap_size" are entries predefined by the CAT compiler, whereas for "fixed_heap" and "heap_size" you can choose other names. The "size in bytes" and the "no. of possibly alloc. segments" define the total heap size as follows:

$$\text{Total heap size} = \sum_{A=0}^{n-1} 2^A * (\text{size in bytes}), \text{ with A=no. of segments.}$$

If "no. of possibly allocated segments" is given as 0 or the parameter is left out, the equation above would be meaningless. In that case the LINKAGE-LOADER functions as if all the commands were omitted; the heap will be expanded dynamically with the need for space.

As a segment can hold 2^{27} bytes, the most commonly used value for "no. of possibly alloc. segments" for a fixed heap size is 1.

The following examples illustrate how you can define that fixed heap size:

```
@LINKAGE-LOADER
NLL: CC Load your program, the C library and the CAT library
NLL: DEFINE-ENTRY fixed_heap 1 d↵
NLL: DATA-REFERENCE fixed_heap rts_fixed_heap d↵
NLL: DEFINE-ENTRY heap_size 100000B d↵
NLL: DATA-REFERENCE heap_size rts_heap_size d↵
NLL: CC The total heap size is 1*100000B = 100000B
NLL: EXIT↵
```

```
@LINKAGE-LOADER
NLL: CC Load your program, the C library and the CAT library
NLL: DEFINE-ENTRY nsegments 2 d↵
NLL: DATA-REFERENCE nsegments rts_fixed_heap d↵
NLL: DEFINE-ENTRY howbig 100000B d↵
NLL: DATA-REFERENCE howbig rts_heap_size d↵
NLL: CC total heap size = 1*100000B + 2*100000B = 300000B
NLL: EXIT↵
```

NOTES

The initial size of the available heap memory can be adjusted when linking the program. Otherwise, if more space than initially allocated is needed, the heap is dynamically expanded by allocation of new segments.

Example of *malloc* and *free* on page 5-7.

Memory Functions

The memory functions operate efficiently on memory areas, which do not necessarily contain null terminated strings; their length is determined by a count parameter. These functions do not check for overflow of any receiving memory area.

FUNCTION memccpy, memchr, memcmp, memcpy, memset;

HEADER FILE #include <memory.h>

DECLARATION

- ```
  char *memccpy (s1, s2, c, n);
  char *s1, *s2;
  int c, n;
  ```
- ```
  char *memchr (s, c, n);
  char *s;
  int c, n;
  ```
- ```
  int *memcmp (s1, s2, n);
  char *s1, *s2;
  int n;
  ```
- ```
  char *memcpy (s1, s2, n);
  char *s1, *s2;
  int n;
  ```
- ```
  char *memset (s, c, n);
  char *s;
  int c, n;
  ```

DESCRIPTION

memccpy — *memccpy* copies bytes from memory area *s2* into *s1*, stopping after the first occurrence of the character *c* has been copied, or after *n* characters have been copied, whichever comes first. It returns a pointer to the character after the copy of *c* in *s1*, or a NULL pointer if *c* was not found in the first *n* characters of *s2*. The copying always proceeds from left to right; so be careful with overlapping memory areas.
  ```

memchr

memchr returns a pointer to the first occurrence of the character *c* in the first *n* characters of the memory area *s*, or a NULL pointer if *c* does not occur in the specified area.

memcmp

memcmp compares the first *n* characters of its parameters. It returns an integer less than, equal to, or greater than zero, according as *s1* is lexicographically less than, equal to, or greater than *s2*.

memcpy

memcpy copies *n* characters from memory area *s2* into *s1* and returns a pointer to *s1*. If *s1* and *s2* are overlapping areas, copying proceeds from right to left, thus assuring that the copying will be done correctly.

memset

memset sets the first *n* characters in memory area *s* to the value of the character *c* and returns a pointer to this area.

You will find an example on page 15-27.

— Global Jumps

The global jump functions allow you to jump from one function to another. They are mainly used for dealing with errors and interrupts encountered in a low-level function of a program.

FUNCTION *setjmp, longjmp*

HEADER FILE **#include** <setjmp.h>

DECLARATION

- ```
  int setjmp (env);
  jmp_buf env;
  ```
- ```
  void longjmp (env, val);
  jmp_buf env;
  int val;
  ```

DESCRIPTION

setjmp

setjmp saves its return address (after the call) in its parameter *env*, whose type *jmp_buf* is defined in the header file *setjmp.h*. For each call of *setjmp* you have to declare an individual variable of type *jmp_buf* in your program. If not invoked by *longjmp*, *setjmp* returns the value 0.

longjmp

longjmp jumps to the address saved by a previous corresponding *setjmp* call simulating a return from *setjmp* with value *val*.

longjmp can never cause *setjmp* to return the value 0. If *longjmp* is called with *val* set to 0, *setjmp* will return the value 1. All accessible data have values as of the time *longjmp* was called.

NOTES

The result of a call to *longjmp* without a previous corresponding call to *setjmp* is undefined.

example

```
#include <setjmp.h>
static jmp_buf env;

static f(i)
int i;
{
  if (i < 5)
    printf ("i = %d\n", i);
    else longjmp (env, 100);
}

main()
{
  int val, i;

  val = setjmp (env);
  printf ("val = %d\n", val);

  if (val == 100) {
    printf ("--- global jump ---\n");
    return 0; }

  for (i=0; i<10; i++)
    f(i);
}
```

This program produces the following output:

```
val = 0
i = 0
i = 1
i = 2
i = 3
i = 4
val = 100
--- global jump ---
```

String Functions

String functions operate on null-terminated strings. They do not check for overflow of any receiving string.

FUNCTION length of a string: *strlen*

HEADER FILE **#include <string.h>**

DECLARATION

- **int** strlen (s);
  char *s;

DESCRIPTION *strlen* returns the number of non-null characters in *s*.

You will find an example of *strlen* on page 13-66.

FUNCTION append a string: *strcat, strncat, strcatn*

HEADER FILE **#include <string.h>**

DECLARATION

- **char** *strcat (s1, s2);
  char *s1, *s2;
- **char** *strncat (s1, s2, n);
  char *s1, *s2; **int** n;
- **char** *strcatn (s1, s2, n);
  char *s1, *s2; **int** n;

DESCRIPTION *strcat* copies string *s2* to the end of string *s1*. Both, *strncat* and *strcatn* copy at most *n* characters. All three functions return a pointer to the null-terminated result.

NOTES The function *strcatn* is implemented for compatibility reasons only.

example

```
main()
{
  char s1[]="Norsk ";
  char s2[]="Data";
  strcat(s1, s2);
  printf("%s", s1);
}
```

FUNCTION copy a string: *strcpy*, *strncpy*, *strcpyn*

HEADER FILE #include <string.h>

DECLARATION

- char *strcpy (s1, s2);
  char *s1, *s2;

- char *strncpy (s1, s2, n);
  char *s1, *s2; **int** n;

- char *strcpyn (s1, s2, n);
  char *s1, *s2; **int** n;

DESCRIPTION *strcpy* copies string *s2* to *s1*, stopping after the null character has been moved. *strncpy* and *strcpyn* copy exactly *n* characters, truncating or null-padding *s2*; the target may not be null-terminated, if the length of *s2* is *n* or more. All three functions return a pointer to *s1*.

NOTES The function *strcpyn* is implemented for compatibility reasons only.

You will find an example of *strncpy* on page 15-61.

FUNCTION

compare two strings:
strcmp, strncmp, strcmpn, strspn, strcspn

HEADER FILE

```
#include <string.h>
```

DECLARATION

- ```
  int strcmp (s1, s2);
  char *s1, *s2;
  ```
- ```
  int strncmp (s1, s2, n);
  char *s1, *s2;  int n;
  ```
- ```
  int strcmpn (s1, s2, n);
  char *s1, *s2;  int n;
  ```
- ```
  int strspn (s1, s2);
  char *s1, *s2;
  ```
- ```
  int strcspn (s1, s2);
  char *s1, *s2;
  ```

DESCRIPTION

strcmp

strcmp compares its parameters and returns an integer greater than, equal to, or less than 0, according as *s1* is lexicographically greater than, equal to, or less than *s2*.

strncmp, strcmpn

strncmp and *strcmpn* make the same comparison as *strcmp* but look at at most *n* characters.

strspn

strspn returns the number of characters in the initial part of *s1* that are also part of *s2*. The comparison stops when the first character in *s1* is reached that cannot be found in *s2*.

strcspn

strcspn returns the number of characters in the initial part of *s1* that cannot be found in *s2*. The comparison stops when the first character in *s1* is reached that can be found in *s2*.

NOTES

The function *strcmpn* is implemented for compatibility reasons only.
  ```

example

```
#include <string.h>

main()
{
  char *s1, *s2;

  s1 = "abaababbcde";  s2 = "ab";
  printf ("result = %d\n", strspn(s1,s2));

  s1 = "axyz";  s2 = "edbac";
  printf ("result = %d\n", strcspn(s1,s2));
}
```

This program produces the following output:

```
result = 8
result = 0
```

FUNCTION — search for a character: *index*, *strchr*, *rindex*, *strrchr*

HEADER FILE — #include <string.h>

DECLARATION

- ```
  char *index(s,c);        char *strchr(s, c);
  char *s, c;              char *s, c;
  ```
- ```
  char *rindex(s,c);       char *strrchr(s,c);
  char *s, c;              char *s, c;
  ```

DESCRIPTION

index, *strchr* — Both, *index* and *strchr* return a pointer to the first occurrence of the character *c* in string *s*.

rindex, *strrchr* — Both, *rindex* and *strrchr* return a pointer to the last occurrence of the character *c* in string *s*.

The return value NULL indicates that *c* does not occur in the string.

NOTES — The function names *strchr* and *strrchr* are implemented for compatibility reasons.

Remember, that the terminating null character is part of a string; this is important, if you want to determine the end position of a string.

example

```
main()
{
  char *first, *last;
  char s[]="Norsk Data";

  printf("%s\n",*s);
  first = index(s, 'a');
  last = rindex(s, 'a');
  printf("First occurence of 'a' in string at position %ld.\n",
                                                    first - s + 1);
  printf("Last occurence of 'a' in string at position %ld.\n",
                                                     last - s + 1);
}
```

FUNCTION search for characters: *strpbrk*

HEADER FILE **#include** <string.h>

DECLARATION

- **char** *strpbrk (s1, s2);
  char *s1, *s2;

DESCRIPTION *strpbrk* returns a pointer to the first occurrence of any character of string *s2* in string *s1*, or NULL if no character from *s2* occurs in *s1*.

example

```
#include <string.h>
#include <stdio.h>
main()
{
  char s1[] = "Norsk Data";
  char s2[] = " ";
  char *ptr;

  if ((ptr = strpbrk (s1, s2)) != NULL) {
    printf("Occurence of s2 in s1 ");
    printf("at position %d.\n", ptr - &s1[0] + 1);
  }
  else printf("s2 does not occur in s1");
}
```

FUNCTION text tokens: *strtok*

HEADER FILE **#include** <string.h>

DECLARATION

- **char** *strtok (s1, s2);
  char *s1, *s2;

DESCRIPTION *strtok* considers the string *s1* to consist of a sequence of text tokens separated by characters from the separator string *s2*. Subsequent calls to *strtok*, operating on the same string *s1*, produce null-terminated substrings of *s1* and return each a pointer to the next following substring.

The first call (with *s1* specified) returns a pointer to the first character of the first token, and will have written a null character immediately after that token. For subsequent calls, which are to work through the string *s1*, NULL must be specified as first parameter.

In this way, *strtok* can keep track of its position in the string between separate calls. Each subsequent call returns a pointer to the first character of the next token and inserts a null character after it.

The separator string *s2* may vary from call to call. When the end of *s1* is reached, a NULL pointer is returned.

example

```
#include <string.h>
#include <stdio.h>
main()
{
  char *ptr;
  char s1[] = "string/split,by*strtok";
  char s2[] ="/*,";
  int i;

  printf("%s\n", s1);
  strtok(s1, s2);
  i = 1;
  printf("Token %d : %s\n", i, s1);
  while ((ptr = strtok(NULL, s2)) != NULL)
  {
    i++;
    printf("Token %d : %s\n", i, ptr);
  }
}
```

— Character Functions

FUNCTION
character conversion:
toupper, tolower, toascii

HEADER FILE
#include <ctype.h>

DECLARATION

- **int** toupper(c)
  int c;
- **int** tolower(c)
  int c;
- **int** toascii(c)
  int c;

DESCRIPTION

toupper, tolower

toupper and *tolower* have a valid parameter range from -1 to 255. If the parameter of *toupper* is a lower case letter, the result is the corresponding upper case letter. If the parameter of *tolower* is a upper case letter, the result is the corresponding lower case letter. All other parameters remain unchanged.

example

```
/* changes lower case letters to uppercase */
/* letters and vice versa */
#include <stdio.h>
#include <ctype.h>
main() {
  int  c;
  for (c=1; c < 128; c++) {
    if (islower(c)) printf("%c ",toupper(c));
    if (c == 91) printf("\n");
    if (isupper(c)) printf("%c ",tolower(c));
  }
 }
```

toascii

toascii returns its parameter with all bits turned off that are not part of the standard ASCII character.

FUNCTION	character classification macros: *isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii*
HEADER FILE	#include <ctype.h>
DECLARATION	All these "functions" are implemented as macros and are called with a parameter of type **char** (*isalpha(c)*, etc).
DESCRIPTION	These macros classify ASCII-coded integer values by table lookup. Each is a predicate returning non-zero for true, zero for false. *isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value EOF (-1).
isalpha	*c* is a letter.
isupper	*c* is an upper case letter.
islower	*c* is a lower case letter.
isdigit	*c* is a digit.
isxdigit	*c* is a hexadecimal digit [0-9],[A-F] or [a-f].
isalnum	*c* is an alphanumeric character.
isspace	*c* is a blank, tab, carriage return, newline or formfeed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character: code 32 (blank) through 126 (tilde).
isgraph	*c* is a printing character, except for blank: code 33 (exclamation mark) through 126 (tilde).
iscntrl	*c* is a delete character (127) or an ordinary control character (less than 32).
isascii	*c* is an ASCII character: code 0 through 127.
NOTES	If the parameter to any of these macros is not in the domain of the function, the result is undefined.

example

```
#include <stdio.h>
#include <ctype.h>
main() {
  int  c;
  char iss[]    = "  x   ";
  char isl[]    = "  x    ";
  char isnots[] = "      ";
  char isnotl[] = "       ";

  printf("      alpha upper lower digit xdigit alnum space punct");
  printf(" print graph cntrl ascii\n");
  for (c=1; c < 128; c++) {
    if ((c > 31) && (c != 127))
      printf("%c %3d ", c, c);
    else printf("  %3d ", c);
    if (isalpha(c)) printf("%s", iss);
    else printf("%s", isnots);
    if (isupper(c)) printf("%s", iss);
    else printf("%s", isnots);
    if (islower(c)) printf("%s", iss);
    else printf("%s", isnots);
    if (isdigit(c)) printf("%s", iss);
    else printf("%s", isnots);
    if (isxdigit(c)) printf("%s", isl);
    else printf("%s", isnotl);
    if (isalnum(c)) printf("%s", iss);
    else printf("%s", isnots);
    if (isspace(c)) printf("%s", iss);
    else printf("%s", isnots);
    if (ispunct(c)) printf("%s", iss);
    else printf("%s", isnots);
    if (isprint(c)) printf("%s", iss);
    else printf("%s", isnots);
    if (isgraph(c)) printf("%s", iss);
    else printf("%s", isnots);
    if (iscntrl(c)) printf("%s", iss);
    else printf("%s", isnots);
    if (isascii(c)) printf("%s\n", iss);
    else printf("%s\n", isnots);
  }
}
```

— Conversion Functions

FUNCTION convert ASCII string to integer: *atoi, atol,strtol*

HEADER FILE #include <string.h>

DECLARATION

```
• int atoi (str);
  char *str;

• long atol (str);
  char *str;

• long strtol (str, ptr, base);
  char *str;
  char **ptr;
  int base;
```

DESCRIPTION

These functions convert a string pointed to by *str* to integer and long integer representation respectively. The first unrecognised character ends the string. The string parameter must obey the syntax rules for decimal integer constants.

strtol

A *base* specification between 2 and 36 is used as the base for conversion. If the *base* equals 16, the *0x* or *0X* will be ignored; leading zeroes after an optional leading sign will be ignored as well.

If the *base* is zero, the string itself determines the base: A leading zero after an optional leading sign indicates an octal number, whereas a leading *0x* or *0X* indicates a hexadecimal number. Otherwise, decimal conversion is used.

If the pointer *ptr* is unequal NULL, the address of the first unrecognised character is returned in **ptr*. Successive calls with *str* set to **++ptr* can be used for example to convert integer numbers, which are collected together in a string and separated by blanks.

If no integer can be formed, **ptr* is set to *str* and zero is returned.

example

```
#include <string.h>
#include <stdio.h>
main() {
  char *str = "23 45 68 11";
  char **ptr;
  long l;

  while ((l = strtol(str, ptr, 0)) && (*ptr != str)) {
    printf("%ld\n", l);
    str = ++*ptr;
  }
}
```

You will find an example of *atoi* on page 13-38.

The following relationship between *strtol* and *atol* resp. *atoi* can be defined:

```
atol (str)

is equivalent to

strtol (str, (char **) NULL, 10)
```

```
atoi (str)

is equivalent to

(int) strtol (str, (char **) NULL, 10)
```

NOTES

In the event of an overflow the maximum value possible to represent will be returned, and *errno* is set to ERANGE.

FUNCTION

convert ASCII string to floating point number: *atof*

HEADER FILE

#include <math.h>

DECLARATION

- **double** atof (nptr);
  char *nptr;

DESCRIPTION

atof converts a string pointed to by *nptr* to a floating point number. The first unrecognised character ends the string. A valid string parameter to *atof* must obey the syntax rules for floating constants (see page 2-8).

NOTES

If the parameter string starts with an unrecognised character, 0 is returned.

FUNCTION convert numbers to strings: *ecvt*, *fcvt*, *gcvt*

HEADER FILE **#include** <math.h>

DECLARATION

- **char** *ecvt (value, ndigit, decpt, sign);
  double value;
  int ndigit, *decpt, *sign;

- **char** *fcvt (value, ndigit, decpt, sign);
  double value;
  int ndigit, *decpt, *sign;

- **char** *gcvt (value, ndigit, buf);
  double value;
  int ndigit;
  char *buf;

DESCRIPTION

ecvt

ecvt converts the specified *value* to a null-terminated string of *ndigit* ASCII digits and returns a pointer to it. This string does not contain the optional decimal point; the position of the decimal point relative to the beginning of the string is stored indirectly through *decpt*. If *decpt* is negative, the decimal point is to the left of the returned digits. The integer pointed to by *sign* is zero, if the result is positive, otherwise non-zero. The low-order digit is rounded.

fcvt

fcvt is identical to *ecvt*, except that the correct digit has been rounded for FORTRAN F-format output of the number of digits specified by *ndigit*, i.e. *ndigit* specifies the number of digits after the (not included) decimal point.

gcvt

gcvt converts the specified *value* to a null-terminated ASCII string in *buf* and returns a pointer to *buf*. It attempts to produce *ndigit* significant digits in FORTRAN F-format. The E-format is used, when the exponent of the value is less than or equal to -4 or if it is greater than *ndigit-1*. For a value with a ten-exponent less than zero (e.g. 0.01) the zero before the decimal point is not significant. The string returned is ready for printing, i.e. it contains a decimal point and optionally a minus sign.

NOTES The return values point to static data whose content is overwritten by each call.

— Mathematical Functions

overflow

On the ND-500 the overflow check is implemented in the hardware part and cannot be switched off. This means, that in the event of an overflow the program will abort. Therefore, we advise you to check the value of *errno* whenever an overflow can be expected.

hardware instructions

For some functions (*exp*, *log*, *log2*, *log10*, *sqrt*, *sin*, *cos*, *tan*, *asin*, *acos*, *atan*) hardware instructions can be taken instead of software functions. They are faster, but have some drawbacks: the calculations are not as exact as with software functions, return values of hardware instructions are not checked and you can't have a pointer to a hardware instruction. Hardware instructions are taken when you include the header file *math.h* and define the following macro:

```
#define HARDWARE 1
```

FUNCTION

absolute integer value: *abs*

HEADER FILE

#include <math.h>

DECLARATION

- **int** abs (i);
  int i;

DESCRIPTION

abs is available as a function and as a macro. When including the header file *math.h* the function will be used, otherwise the macro, which is faster. Both return the absolute value of its integer operand.

When calling *abs* with the most negative integer value -2147483648 as parameter, the macro returns this value unchanged, whereas the function returns the maximum positive value 2147483647 and sets *errno* to ERANGE.

FUNCTION — exponent, logarithm, power and square root: *exp, log, log2, log10, pow, ipow, dpow, sqrt*

HEADER FILE — #include <math.h>

DECLARATION

- ```
  double exp(x);
  double x;
  ```
- ```
  double log(x);
  double x;
  ```
- ```
  double log2(x);
  double x;
  ```
- ```
  double log10(x);
  double x;
  ```
- ```
  double pow(x,y);
  double x,y;
  ```
- ```
  int ipow(i,j);
  int i,j;
  ```
- ```
  double dpow(x,i);
  double x;
  int i;
  ```
- ```
  double sqrt(x);
  double x;
  ```

DESCRIPTION

exp — *exp* returns the result of e^x, where *e=2.718281828...* . If the result is too large to be represented, a huge value will be returned and *errno* is set to ERANGE.

log — *log* returns the natural logarithm of *x* (base *e*)). If the parameter is equal to or less than zero, a very small value is returned and *errno* is set to EDOM.

log2 — *log2* returns the logarithm to the base 2. If the parameter is equal to or less than zero, a very small value is returned and *errno* is set to EDOM.

log10 — *log10* returns the logarithm to the base 10. If the parameter is equal to or less than zero, a very small value is returned and *errno* is set to EDOM.

pow — *pow* returns the result of x^y, where the parameters as well as the result are floating point values. If the first parameter is negative and the second is not an integer, zero is returned and *errno* is set to EDOM. A call to *pow* with both parameters equal to zero returns zero as well, but does not cause an error.

ipow — *ipow* returns the result of i^j;, where the parameters as well as the result are integer values.

dpow — *dpow* returns the result of x^i;, where the base and the result are floating point values and the exponent is an integer value.

sqrt — *sqrt* returns the square root of x. A negative parameter causes *sqrt* to return zero and *errno* to be set to EDOM.

FUNCTION generate random values: *rand, srand*

HEADER FILE #include <math.h>

DECLARATION

- int rand();
- void srand(i);
  int i;

DESCRIPTION

rand — *rand* uses a multiplicative congruential random number generator to return successive pseudo-random numbers in the range from 0 to the highest int value.

srand — The parameter of *srand* sets a starting point for a new sequence of pseudo-random numbers to be returned by subsequent calls to *rand*. As such it can be used to reinitialise the generator.

FUNCTION

trigonometric functions:
sin, cos, tan, asin, acos, atan, atan2

HEADER FILE

#include <math.h>

DECLARATION

- **double** sin(x);
  double x;
- **double** cos(x);
  double x;
- **double** tan(x);
  double x;
- **double** asin(x);
  double x;
- **double** acos(x);
  double x;
- **double** atan(x);
  double x;
- **double** atan2(x,y);
  double x, y;

DESCRIPTION

sin, cos, tan

sin, *cos* and *tan* return the trigonometric result of radian parameters. The value of the parameter should ensure a meaningful result.

If the cosine of the parameter equals zero, *tan* returns a huge value and *errno* is set to ERANGE.

asin

asin returns the arc sine of x in the range [-π/2..+π/2].

acos

acos returns the arc cosine of x in the range [0..π].

atan

atan returns the arc tangent of x in the range [-π/2..+π/2].

atan

atan2 returns the arc tangent of x/y in the range [-π..π].

NOTES

Parameters out of the range [-1..+1] cause *asin* and *acos* to return zero; *errno* is set to EDOM.

Parameters to *tan* should not be greater than the maximum integer value (2147483647). The value of *tan* at its singular points is too large to be represented; *errno* is set to ERANGE.

atan2 returns zero and *errno* is set to EDOM, if both parameters are zero.

FUNCTION

hyperbolic functions: *sinh*, *cosh*, *tanh*

HEADER FILE

#include <math.h>

DECLARATION

- **double** sinh(x);
  double x;

- **double** cosh(x);
  double x;

- **double** tanh(x);
  double x;

DESCRIPTION

These functions compute the designated hyperbolic results for real parameters.

NOTES

In the event of an overflow *sinh* and *cosh* return the maximum value possible to be represented (of the appropriate sign), and *errno* is set to ERANGE.

FUNCTION bessel functions: *j0, j1, jn, y0, y1, yn*

HEADER FILE **#include** <math.h>

DECLARATION

- ```
  double j0(x);
  double x;
  ```
- ```
  double j1(x);
  double x;
  ```
- ```
  double jn(n,x);
  int n;
  double x;
  ```
- ```
  double y0(x);
  double x;
  ```
- ```
  double y1(x);
  double x;
  ```
- ```
  double jn(n,x);
  int n;
  double x;
  ```

DESCRIPTION These functions calculate Bessel functions of the first and second kinds for real parameters and integer orders.

NOTES Negative parameters cause *y0*, *y1* and *yn* to return a large negative value and set *errno* to EDOM.

FUNCTIONS	error function of an double argument: *erf*,*erfc*
HEADER FILE	#include <math.h>
DECLARATION	● double erf(d); double d; double erfc(d); double d;
DESCRRIPTION	The function *erf* returns the error function of its double argument. *Erfc* returns 1.0 - erf(argument). *Erfc* is provided because of the extreme loss of accurary when erf is called for large x and the result is subtracted from 1 (for example: with x = 10 :12 places are lost). There are no error returns. Coefficients for large x are #5667 from Hart & Cheney (18.72D).

FUNCTION

absolute floating point value, floor, ceil, fmod:
fabs, floor, ceil, fmod

HEADER FILE

#include <math.h>

DECLARATION

- **double** fabs (x);
  double x;
- **double** floor (x);
  double x;
- **double** ceil (x);
  double x;
- **double** fmod (x, y);
  double x, y;

DESCRIPTION

fabs — *fabs* returns the absolute value of its floating operand.

floor — *floor* returns the largest integer not greater than *x*, represented as a double.

ceil — *ceil* returns the smallest integer not less than *x*, represented as a double.

fmod — *fmod* returns the floating point remainder of *x/y*. If *y* equals zero, *x* is returned; otherwise, a value with the same sign as *x* is returned, such that

x = i * y + fmod

where *i* is an integer and *fmod* is less than *y* (absolute values).

FUNCTION logarithmic gamma function: *gamma*

HEADER FILE #include <math.h>

DECLARATION

```
• double gamma(x);
  double x;
```

DESCRIPTION *gamma* returns $\ln |G(|x|)|$ where G denotes the gamma function. The sign of $G(|x|)$ is returned in the external integer *signgam*.

NOTES For negative integer parameters the maximum integer value is returned and *errno* is set to EDOM.

example The following C program may be used to calculate G:

```
#include <math.h>

double G(x)
double x;
{
  double y;
  y = gamma(x);
  if (y > 88.0)
     error();
  y = exp(y);
  if(signgam)
     y = -y;
  return y;
}
```

There should be a positive indication of *error*.

FUNCTION

split into mantissa and exponent:
frexp, ldexp, modf, dint, dintr

HEADER FILE

#include <math.h>

DECLARATION

- **double** frexp(value, eptr);
  double value;
  int *eptr;

- **double** ldexp(value, exp);
  double value;
  int exp;

- **double** modf(value, iptr);
  double value, *iptr;

- **double** dint(value);
  double value;

- **double** dintr(value);
  double value;

DESCRIPTION

frexp

frexp returns the mantissa of a **double** *value* as a **double** quantity, which is less than 1. Its parameter *eptr* points to an integer *n* such that $value = x*2^{n}$. If *value* equals zero, the return value as well as the integer *n* will be zero.

ldexp

ldexp returns the result of $value*2^{exp}$.

modf

modf returns the positive fractional part of *value* and stores the integer part indirectly through *iptr*.

dint, dintr

dint and *dintr* return the integer part of their **double** parameter represented as a double; if necessary *dintr* rounds the result.

FUNCTION euclidean distance: *hypot, cabs*

HEADER FILE #include <math.h>

DECLARATION

- **double** hypot(x,y);
  double x,y;

- **double** cabs(z);
  struct {**double** x,y;} z;

DESCRIPTION *hypot* and *cabs* return the result of *sqrt (x*x + y*y)* taking precautions against unwarranted overflows.

— Other Standard Functions

FUNCTION execute a program: *execvp*, *execlp*

DECLARATION

- **int** execvp (name, argv);
  char *name, *argv[];
- **int** execlp (name, arg0, arg1,...,argn, 0);
  char *name, *arg0, *arg1,...,*argn;

DESCRIPTION Both, *execvp* and *execlp* call the basic function *execve* (see page 13-27).

execvp *execvp* works exactly the same as *execve*.

execlp *execlp* works as *execve*, with the only difference that the parameters are given explicitly. The first parameter *arg0* is ignored. A maximum of 8 actual parameters (*arg1* up to *arg8*) is allowed. The end of the parameter list is indicated by the last parameter of *execlp* which is 0.

FUNCTION handle variable argument list

HEADER FILE #include <varargs.h>

DECLARATION

```
any_function(va_alist)   /* I.e. your routine, using a
                            variable argument list.   */
va_dcl                   /* Without semicolon!!       */
va_list pvar;
va_start(pvar);
type some_variable;
some_variable = va_arg(pvar, type);
va_end(pvar);
```

OR if "any_function" calls "another_func" passing the parameter list to it after having already started the parameter list:

```
another_func(args)
va_list args;
```

DESCRIPTION A machine and compiler dependent set of macros defined in the header file varargs.h provide facilities of writing portable procedures that accept variable argument lists. Routines with variable argument lists (as f.e.

the standard routines for formatted I/O printf and scanf) are nonportable without varargs because different machines/compilers use different argument passing conventions.

Under SINTRAN va_alist is defined as the list of arguments. In order to provide enough space on the stack to hold all arguments it is defined with the maximum number of possible arguments (32 under SINTRAN). Va_alist is used in a routine header to declare a variable argument list, but as under SINTRAN va_alist is already defined in varargs.h you may not redefine its value which may be possible on other systems.

Va_dcl is a declaration for va_alist. No semicolon must follow va_dcl.

Va_list is a type definition for the variable used to scan the list. At least one variable of type va_list must be declared. In the declaration above pvar is used.

Va_start(pvar) is called to make the variable pvar of type va_list point to the beginning of the argument list.

Va_arg(pvar, type) will return the next argument in the list pointed to by pvar. "Type" is the type the argument is expected to be. Within subsequent calls of va_arg different types may be given. It is up to the routine to "know" what type is expected, since this cannot be determined at runtime. Up to now float, double, struct and union are passed by reference. This implies that care must be taken when these types occur in the argument list. If "type" of the va_arg function is expected to be of one of these types "type*" has to be the second parameter of the call to va_arg. Sources containing these types in a variable parameter list are not portable.

In future versions, the passing conventions of these types may be changed to gain more portability. A look into the header file ensures you to be programming with right passing convention in mind.

Va_end(pvar) is used to finish the scanning of the argument list pointed to by pvar.

Multiple scannings through argument lists, each enclosed by va_start ... va_end, are possible.

The argument list (or its remainder) may be passed to another function using a variable of type va_list. In this case a call to va_arg in the subroutine will scan the list with respect to the caller as well.

example

A possible implementation of scanf:

```
#include <stdio.h>
#include <varargs.h>
int scanf(format, va_alist)
char *format;
va_dcl
{
 va_alist lptr;
 va_start(lptr);
 return(_doscan(stdin, format, lptr));
}

int _doscan(iop, fmt, args)
FILE *iop;
char *fmt;
va_list args;
{
 int argument,assign_no,ok;
 .....
 /* _doscan now does all the formatting work.  */
 /* If fmt tells that an argument is expected, */
 /* it calls another subroutine to handle the  */
 /* argument:                                  */
 if (argument) {
  ok=handle_argument(fmt^, iop, &args);
  if (ok) assign_no++;
 }
 .....
 /* continue with the formatting */
 .....
 return(assign_no);
}

int handle_argument(c, iop, lptrptr)
char c;
FILE *iop;
va_list *lptrptr;
{
 /* in a very simple form: */
 double dval;
 int ival,is_right=1;
 char *cval;
 .......
 /* As va_arg is implemented as a macro, using       */
 /* this "function" the other way around is also     */
 /* possible. Dval and ival are the numbers from the */
 /* character input which have to be converted to    */
 /* double or int respectively. They are converted   */
 /* to the appropriate pointer values via va_arg :   */
 if (c == 'e') *va_arg(*lptrptr, double *) = dval;
 if (c == 'd') *va_arg(*lptrptr, int *) = ival;
 if (c == 's') cval = va_arg(*lptrptr, char *);
 ......
 return(is_right);
}
```

NOTES

The routine calling va_arg has to determine the number of arguments. The scanf and printf routines for example do this from the given format string. Other routines calling var_arg may have a programmed end of scanning of the argument list as for instance a zero value or a zero pointer.

FUNCTION

execute system command: *system*

DECLARATION

- **int** system (string);
  char *string;

DESCRIPTION

If the *string* represents a valid SINTRAN command, it will be executed by the SINTRAN command processor.

RETURN VALUE

This function always returns 0, regardless whether the SINTRAN command was executed successfully or not.

NOTES

If the command executed by system is also available in the monitor call interface, you should use that one instead. Using monitor calls offers the advantage that you can check the success of the operation by calling the function ***errcode*** afterwards (see page 15-3).

example

```
main()
{
  system("list-files, :c");
}
```

FUNCTION

get processing time: *times*

HEADER FILE

```
#include <types.h>
#include <times.h>
```

DECLARATION

```
• int times(buffer);
  struct tms *buffer;
```

DESCRIPTION

The function *times* returns the CPU time used in the current process in 1/60 seconds. As there are no child processes under SINTRAN, the result will be stored in the first component (tms_utime) of the structure *tms* only which is declared in the header files as:

```
struct tms {
       long tms_utime;
       long tms_stime;
       long tms_cutime;
       long tms_cstime;
       };
```

The other components of the structure *tms* contain zeros.

example

```
#include <types.h>
#include <times.h>
#include <stdio.h>

main()
{
  struct tms buffer;
  int i;

  for (i=0; i<3000000; i++);;
  times(&buffer);
  printf ("utime  = %d\n",buffer.tms_utime);
  printf ("stime  = %d\n",buffer.tms_stime);
  printf ("cutime = %d\n",buffer.tms_cutime);
  printf ("cstime = %d\n",buffer.tms_cstime);
}
```

The output could look like:

```
utime  = 288
stime  = 0
cutime = 0
cstime = 0
```

FUNCTION get time in seconds: *time*, *ftime*

HEADER FILE

```
#include <types.h>
#include <timeb.h>
```

DECLARATION

- ```
  long time(tloc);
  long *tloc;
  ```
- ```
  long ftime(tp);
  struct timeb *tp;
  ```

DESCRIPTION Under SINTRAN, the time functions do not cover the same functionality as they do under UNIX. Especially, the time zone is always considered to be Greenwich.

time The function *time* returns the time since 1st January 1970, 00:00:00 hrs, measured in seconds. If *tloc* is unequal zero, the return value is also stored in the place to which *tloc* points.

ftime The function *ftime* fills in a structure pointed to by its parameter. This structure is declared as:

```
struct timeb {
        long time;
        unsigned short millitm;
        int timezone;
        int dstflag;
        };
```

The component *time* will contain the time since 1st January 1970, 00:00:00 hrs, measured in seconds; *millitm* will contain the milliseconds of the current second. The other two values will always be set to zero.

example

```
#include <types.h>
#include <times.h>
#include <stdio.h>

main()
{
  struct timeb tpr;
  ftime (&tp);
  printf ("time     = %d\n",tp.time);
  printf ("millitm  = %d\n",tp.millitm);
  printf ("timezone = %d\n",tp.timezone);
  printf ("dstflag  = %d\n",tp.dstflag);
}
```

The output could look like:

```
time     = 538234903
millitm  = 320
timezone = 0
dstflag  = 0
```

NOTES

Under SINTRAN *ftime* returns the same values as the basic function *gettimeofday*.

FUNCTION

convert date and time into ASCII:
ctime, localtime, gmtime, asctime

HEADER FILE

```
#include <time.h>
```

DECLARATION

- ```
  char *ctime(clock);
  long *clock;
  ```
- ```
  struct tm *localtime(clock);
  long *clock;
  ```
- ```
  struct tm *gmtime(clock);
  long *clock;
  ```
- ```
  char *asctime(tm);
  struct tm *tm;
  ```

DESCRIPTION

ctime

ctime converts a time value measured in seconds into a string containing time and date information. The input parameter may be the result of a call to *gettimeofday*, *time* or *ftime* (see pages 13-30 and 13-106). It returns a pointer to a string of a fixed length of 26 characters, including a carriage return and the terminating null character, e.g.:

```
"Mon Aug 21 11:03:52 1989\n\0"
```

localtime, gmtime

Under SINTRAN *localtime* and *gmtime* are identical. Like *ctime*, they convert a time value measured in seconds into date and time information. But, instead of returning a pointer to a string, they return a pointer to a structure *tm* which contains the broken-down information:

```
struct tm {
       int tm_sec;    /* 0..59 */
       int tm_min;    /* 0..59 */
       int tm_hour;   /* 0..23 */
       int tm_mday;   /* 1..31 */
       int tm_mon;    /* 0 = January              */
       int tm_year;   /* 0..99  - year since 1900 */
       int tm_wday;   /* 0 = Sunday               */
       int tm_yday;   /* 0..365 - day of year     */
       int tm_isdst;  /* always 0                 */
       };
```

asctime

asctime takes a pointer to a structure of type *tm* (as returned by *localtime* or *gmtime*) and returns a pointer to a string with the same layout as in *ctime*.

FUNCTION

create a unique file name: *mktemp*

DECLARATION

- **char** *mktemp(template);
  char *template;

DESCRIPTION

The function *mktemp* tries to combine the character string *template* with the user's process identification to a unique user-dependent file name.

The *template* should contain at least six *x* or *X* (before the file type), which will be replaced by a lower case letter (beginning with *a*) and the user's process identification. If this string represents a non-existing file name, a pointer to it is returned. Otherwise, the same is tried for the next lower case letters, until a unique file name is found. If no sequence of six *x* or *X* can be found and the parameter string itself is a non-existing file name, a pointer to this string will be returned.

Otherwise, if no unique file name can be created, *mktemp* returns a pointer to an empty string.

example

```
Template = "zxxxxxz"       →  mktemp = "zxxxxxz"
Template = "XXXXXXX"       →  mktemp = "Xa00049"
Template = "XXXXXX:XCOM"   →  mktemp = "a00049:XCOM"
```

FUNCTION swap bytes: *swab*

DECLARATION

```
• void swab(source, dest, nbytes);
  char *source, *dest;
  int nbytes;
```

DESCRIPTION The function *swab* enables transfer of binary data-bet ween IBM-compatible and ND-compatible machines (like DEC). It copies *nbytes* from the array pointed to by *source* to the array pointed to by *dest*, exchanging adjacent even and odd bytes (MSB and LSB). *nbytes* should be a positive and even number. If *nbytes* is an odd, positive number, the value *nbytes-1* will be taken instead. If *nbytes* is negative, nothing will be copied.

NOTES This function should only be used for 2-byte integers.

FUNCTION

find if file is a terminal: *ttyname*, *isatty*, *ttyslot*

DECLARATION

- **char** *ttyname(fn);
  int fn;
- **int** isatty(fn);
  int fn;
- **int** ttyslot;

DESCRIPTION

ttyname, *isatty*

The functions *ttyname* and *isatty* are only meaningful to determine if the standard output (*fn* = 1) or input file (*fn* = 0) is associated to your terminal. (When running your program as an interactive process, both are usually associated with your terminal.) If the file number is associated with a terminal, *ttyname* returns a pointer to the name of the terminal; otherwise, a NULL pointer is returned. The names of the standard input and output files are usually defined as "terminal".

isatty returns 1 if *fn* is associated with a terminal, otherwise 0.

ttyslot

ttyslot returns the SINTRAN file number of the standard output, if it is connected to a terminal. Otherwise, 0 is returned.

FUNCTION get login name: *getlogin*

DECLARATION

- char *getlogin();

DESCRIPTION The function *getlogin* returns the name of the user currently logged in. You will find an example on page 13-32.

FUNCTION suspend program execution: *sleep*

DECLARATION

- void sleep(no_of_seconds);
  unsigned no_of_seconds;

DESCRIPTION The function *sleep* suspends program execution for *no_of_seconds* seconds.

FUNCTION terminate program: *exit*, *_exit*

DECLARATION

- void exit (status);
  int status;

- void _exit (status);
  int status;

DESCRIPTION

exit *exit* closes all open files and terminates the calling program.

_exit *_exit* terminates the calling program, but does not close open files.

```
/* Redirection of stdin and stdout to another terminal.                */
/* BEFORE STARTING to try if this program will work, inspect,          */
/* whether there is a terminal with device number 52 that you may      */
/* use. The terminal must not be logged in!                            */
/* If there isn't, look for another terminal you may use and ex-       */
/* change the "52" in the set-peripheral-file command.                 */
/* Before starting the executable program:                             */
/* 1. check via @fi-sta (sys)myperiph,,, that there is a peripheral */
/*    file named "myperiph", whose device no. is 000064 (dec. 52).  */
/* 2. If there isn't a peripheral file "myperiph" under SYSTEM,        */
/*    create one with "@set-peri-file "myperiph" 52d" under SYSTEM. */
/* 3. Now you may start the program.                                    */
/* 4. If anyone else is logged in on terminal 52, you will get:        */
/*    "freopen stdout: Access permission denied, SINTRAN error 98"  */
/* Calls of: freopen, perror, fprintf, scanf, setbuf, break_mode,      */
/*           echo_mode, strlen, fclose, exit                           */
#include <stdio.h>
#include <fcntl.h>
main()
{
 int i, j;
 char line[256];
 FILE *in, *out;
 if ((out=freopen("myperiph", "r+", stdout))==NULL) {
    perror("freopen stdout");
    exit(1);
 }
 setbuf (stdout, NULL);
 if ((in=freopen("myperiph", "r", stdin))==NULL) {
    perror("freopen stdin");
    exit(1);
 }
 setbuf (stdin,NULL);                              /* not buffered at all */
 echo_mode(fileno(in),1);         /* echo on all except control chars */
 break_mode(fileno(in),1);             /* break only on control chars */
 if ((i=printf("Hello Terminal!!!\n")) != 18) {
    fprintf(stderr,"result of 1. printf is not 18, but %2d\n",i);
    perror("printf");
    exit(1);
 }
 if ((i=printf("(now some input please..)\n")) != 26) {
    fprintf(stderr,"result of 2. printf is not 26, but %2d\n",i);
    perror("printf");
    exit(1);
 }
     /* with the following single call of scanf, only one string is */
     /* read, i.e. input from the other terminal also ends with a   */
     /* typed space. If you want to allow more input, you have to   */
     /* program a scanf-loop.                                        */
 if ((i=scanf("%s",line)) != 1)
    fprintf(stderr,"result of scanf should be 1, but is %1d\n",i);
 j = strlen(line);
                                        /* repeat the input as output: */
 if ((i=printf("%s\n",line)) != (j+1))
    fprintf(stderr, "result of printf is not %3d, but %3d\n", (j+1), i);
 fclose(in);
```

```
  if ((i=printf("This was it! Good bye.\n")) != 23) {
     fprintf(stderr, "result of printf is not 23, but %2d\n", i);
     perror("printf");
  }
  fclose(out);
}
```

Chapter 14

Language interfacing

In the following chapter possible ways to interface C main programs and modules written in FORTRAN, PLANC and PASCAL are described. For every data type there is a complete example how to interface using global variables (export / import) and via parameter lists, if possible. In each example there is a variable which is defined in the C main program and output in the "foreign language" module and vice versa. Every example has been tested. A complete mode file to compile and link the programs is enclosed.

Variable sizes in different languages

When mixing modules of different languages you need to know the exact format of the variables. The following table describes the amount of memory allocated by the different variable types.

data-types ND-500	PASCAL	PLANC	FORTRAN	C
pointer	4 bytes	4 bytes	4 bytes	4 bytes
one-byte units - signed - unsigned	 (subrange) byte (char)	 INTEGER1 BYTE	 INTEGER*1 CHARACTER	 char unsigned char
two-bytes units - signed - unsigned	 integer2 byte2	 INTEGER2 RANGE	 INTEGER*2 ---	 short int unsigned short int
four-byte units - signed - unsigned	 integer ---	 INTEGER4 RANGE	 INTEGER*4 ---	 long int unsigned long int
boolean - one byte - two bytes - four bytes	 boolean --- ---	 BOOLEAN1 BOOLEAN2 BOOLEAN4	 LOGICAL*1 LOGICAL*2 LOGICAL*4	 (implemented by integers)
enumeration	implemented by subrange 0..n	ENUMERATION 4 bytes	---	enumeration 4 bytes
subrange types	1)		---	---
32-bit float 64-bit float	real (option r2) real (option r4)	REAL REAL8	REAL*4 REAL*8	float double
sets	SET OF type 2)	TYPE SET 0 .. ub ≤ 255 ub div 8 bytes	---	(implemented by integers)
strings	3)	BYTES BYTE ARRAY PACK	CHARACTER*n n bytes	char[n] n bytes

notes

1) The length of a subrange type 'lb .. ub' is calculated as follows:

(lb >= 0) and (ub < 256)	→	1 byte
(lb >= -128) and (ub < 128)	→	1 byte
(lb >= 0) and (ub < 2^{16})	→	2 bytes
(lb >= -2^{15}) and (ub < 2^{15})	→	2 bytes
others	→	4 bytes

The one-, two- and four byte units are subrange types.

2) The length of a value of the type 'SET OF lb .. ub' is calculated as follows:

```
lb : rounded down to the nearest multiple of 8
ub : rounded up to the nearest multiple of 8
if (ub - lb == 24)
  length = 4 bytes;
  else length = (ub - lb) / 8 bytes.
```

The set bounds must fit in the range -2^{18} .. $2^{18} - 1$.

3) packed array of char.

General rules

To avoid difficulties please observe the following rules:

compiler version

- Make sure that your compiler version is not older than the one decribed here. With older versions there may be incompatibilities. All C programs were compiled with the compiler version A06.

variable types

- Specify the variable types exactly, if you are not quite sure about default values. For instance specify REAL*8 instead of REAL in a FORTRAN subroutine.

variable declaration

When exporting/importing variables from/to a C program, the variables in the C program have to be declared above the main block. Imported variables have to be declared as extern.

stack

If you are trying to use C modules in a program written for example in COBOL, PLANC or FORTRAN you can get problems with the stack. In this case you can define a stack for the C module with *initstack*. For an example see page 10-6.

Interfacing C and FORTRAN

If C and FORTRAN are mixed, we recommend to use a C main program as, unlike FORTRAN, C needs to initialise a run-time stack and a heap.

compiler version

All FORTRAN subroutines were compiled with the K02 version of the FORTRAN-500 compiler.

C compiler option

If you export/import variables the C program has to be compiled with the compiler option ic set to ic+ (externals as common) which is default.

trap handlers

Mixing C and FORTRAN can raise problems with runtime exceptions.

FORTRAN versions before K02

At least for FORTRAN compiler versions before K02 it is assumed that some trap handlers (integer overflow, descriptor range etc.) are disabled. In C all traps are enabled by default in order to catch as many errors as possible. Before calling a FORTRAN subroutine it is necessary to disable the trap handlers corresponding to the variables used in the C main program with the calls

```
clte(9)  disable integer overflow trap handler
clte(13) disable floating underflow trap handler
clte(14) disable floating overflow trap handler
clte(25) disable descriptor range trap handler.
```

After return the traps should be enabled with the calls

```
sete(9)  enable integer overflow trap handler
sete(13) etc..
```

library routine names

There are some runtime system routines in the C and FORTRAN libraries with the same name but a different functionality. When using these routines in the C module and in the FORTRAN module the linker will satisfy these external references with the wrong library routine.

Solution: first load all C modules and libraries. Kill the entries with the same name in the C libraries with the kill-entries command. After that load the FORTRAN modules and libraries.

export/import

Variables being exchanged in the common block have to be declared as common in the FORTRAN subroutine. The names of the FORTRAN common blocks correspond to the names of the variables in the C program.

parameter list

Parameters have to be exchanged by reference. In C the address operator & has to be used.

Export / import of integer variables

```
int c_int;
extern int ftn_int;

main() /* C - FORTRAN, export/import of Integer */
{
  c_int = 1;
  ftn_sub();
  printf("\nC-main\n");
  printf("FORTRAN-integer : %d\n", ftn_int);
}


C      FP-I-1 for ND-500
       SUBROUTINE FTN_sub
       COMMON /c_int/c_var
       INTEGER*4 c_var
       COMMON /ftn_int/ ftn_var
       Integer*4 ftn_var
C
100    FORMAT (X,'C-Integer : ',I2,/)
       WRITE (1,*) 'FORTRAN subroutine'
       WRITE (1,100) c_var
C
       ftn_var = 2
       RETURN
       END
```

Integer variables as parameters

```
main() /* C - FORTRAN, integer as parameter */
{
  int c_int;
  int ftn_int;

  c_int = 1;
  ftn_sub(&c_int, &ftn_int);
  printf("\nC-main\n");
  printf("FORTRAN-integer : %d\n", ftn_int);
}

C      FP-I-2 for ND-500
       SUBROUTINE FTN_sub(C_int, FTN_int)
       INTEGER*4 C_int, FTN_int
100    FORMAT (X,'C-Integer : ',I2,/)
       WRITE (1,*) 'FORTRAN-subroutine'
       WRITE (1,100) C_int
       FTN_int = 2
       RETURN
       END
```

Export / import of real variables

```
float c_float;
double c_double;
extern float ftn_float;
extern double ftn_double;

main() /* C - FORTRAN export/import of Real */
{
  c_float = 1.23;
  c_double = 2.46;
  ftn_sub();
  printf("\nC-main\n");
  printf("FORTRAN-float  : %8.2f\n", ftn_float);
  printf("FORTRAN-double : %8.2f\n", ftn_double);
}

C      FP-R-1 for ND-500
       SUBROUTINE FTN_sub
       COMMON /c_float/c_var1
       REAL*4 c_var1
       COMMON /c_double/c_var2
       REAL*8 c_var2
       COMMON /ftn_float/ftn_var1
       REAL*4 ftn_var1
       COMMON /ftn_double/ftn_var2
       REAL*8 ftn_var2
C
100    FORMAT (X,'C-float  : ',F8.2)
110    FORMAT (X,'C-double : ',F8.2,/)
       WRITE (1,*) 'FORTRAN subroutine'
       WRITE (1,100) c_var1
       WRITE (1,110) c_var2
C
       ftn_var1 = 9.84
       ftn_var2 = 19.68
       RETURN
       END
```

Real variables as parameters

```
main() /* C - FORTRAN  real as parameters */
{
  float c_float;
  double c_double;
  float ftn_float;
  double ftn_double;

  c_float = 1.23;
  c_double = 2.46;
  ftn_sub(&c_float, &c_double, &ftn_float, &ftn_double);
  printf("\nC-main\n");
  printf("FORTRAN-float  : %8.2f\n", ftn_float);
  printf("FORTRAN-double : %8.2f\n", ftn_double);
}

C     FP-R-2 for ND-500
      SUBROUTINE FTN_sub(c_float, c_double, ftn_float, ftn_double)
      REAL*4 c_float, ftn_float
      REAL*8 c_double, ftn_double
C
100   FORMAT (X,'C-float  : ',F8.2)
110   FORMAT (X,'C-double : ',F8.2,/)
      WRITE (1,*) 'FORTRAN subroutine'
      WRITE (1,100) c_float
      WRITE (1,110) c_double
C
      ftn_float = 9.84
      ftn_double = 19.68
      RETURN
      END
```

— Export / import of integer arrays

```
int c_int[5];
extern int ftn_int[5];

main() /* C - FORTRAN  export/import of integer array */
{
  short i;

  for (i = 0; i < 5; ++i)
    c_int[i] = 2 * i;
  ftn_sub();
  printf("\nC-main\n");
  printf("FORTRAN-integer : ");
  for (i = 0; i < 5; i++)
    printf("%2d ", ftn_int[i]);
  printf("\n");
}
```

```
C     FP-IARR-1
      SUBROUTINE FTN_sub
      COMMON /c_int/c_var
      INTEGER*4 c_var(0:4)
      INTEGER*2 I
      COMMON /ftn_int/ ftn_var
      INTEGER*4 ftn_var(0:4)
C
100   FORMAT (X,'C-Integer-Array : ',5(I2,x),/)
      WRITE (1,*) 'FORTRAN-subroutine'
      WRITE (1,100) c_var
      DO 200 I=0,4
        FTN_var(i) = i * i
200   Continue
      RETURN
      END
```

Integer arrays as parameters

When exchanging integer arrays via parameter lists, the arrays have to be passed by reference. As arrays are handled via address by C anyway you don't have to specify the address operator.

```
main() /* C - FORTRAN  integer array as parameter */
{
  int c_int[5];
  int ftn_int[5];
  short i;

  for (i = 0; i < 5; ++i)
    c_int[i] = 2 * i;
  ftn_sub(c_int, ftn_int);
  printf("\nC-main\n");
  printf("FORTRAN-integer : ");
  for (i = 0; i < 5; i++)
    printf("%2d ", ftn_int[i]);
  printf("\n");
}
```

```
C      FP-AI-2 for ND500
       SUBROUTINE FTN_sub(c_int, ftn_int)
       INTEGER*4 c_int(0:4)
       INTEGER*2 I
       INTEGER*4 ftn_int(0:4)
C
100    FORMAT (X,'C-Integer-Array : ',5(I2,x),/)
       WRITE (1,*) 'FORTRAN-subroutine'
       WRITE (1,100) c_int
       DO 200 I=0,4
         FTN_int(i) = i * i
200    Continue
       RETURN
       END
```

Export / import of char arrays

```
char c_chars[12] = "From C";
extern char ftn_chars[12];

main() /* C - FORTRAN  export/import of character array */
{
  ftn_sub();
  printf("\nC-main\n");
  printf("FORTRAN-char-array : ");
  printf("%s \n", ftn_chars);
}


C      FP-C-1 for ND500
       subroutine ftn_sub
       COMMON /c_chars/c_var
       CHARACTER*12 c_var
       COMMON /ftn_chars/ftn_var
       CHARACTER*12 ftn_var
C
100    FORMAT (X,'C-char-array : ',A12,/)
       WRITE (1,*) 'FORTRAN-subroutine'
       Write(1,100) c_var
C
       ftn_var = 'From FORTRAN'
       RETURN
       END
```

Char arrays as parameters

When exchanging char arrays via parameter list, you have to declare a string descriptor in your C main program. This descriptor is generated by the FORTRAN compiler automatically. The type of the descriptor is struct consisting of the size (i.e. number of elements) and the address of the array. As C passes structs by reference you do not have to specify an address operator for the parameters.

```
main()  /* C - FORTRAN  char array as parameter */
{
  struct {  int length;
            char *start;
         }  str_desc1, str_desc2;

  char c_chars[7] = "From C";
  char ftn_chars[12];

  str_desc1.start = &c_chars;
  str_desc1.length = strlen(c_chars);

  str_desc2.start = &ftn_chars;
  str_desc2.length = strlen(ftn_chars);

  clte(25); /* switch off trap handler */
            /* only for FORTRAN versions < K02 */
  ftn_sub(str_desc1, str_desc2);
  sete(25);       /* switch on trap handler */

  printf("\n\nC-main\n");
  printf("FORTRAN-char-array : ");
  printf("%s \n", ftn_chars);
}
```

```
C     FP-C-2
      subroutine ftn_sub(c_chars, ftn_chars)
      CHARACTER*7 c_chars
      CHARACTER*12 ftn_chars
C
100   FORMAT (X,'C-char-array : ',A12,/)
      WRITE (1,*) 'FORTRAN-subroutine'
      Write(1,100) c_chars
C
      ftn_chars = 'From FORTRAN'
      RETURN
      END
```

Export / import of structs

```
struct {  int    c_int;
          int    ftn_int;
          double c_float;
          double ftn_float;
          int    c_iarr[5];
          int    ftn_iarr[5];
          char   c_chars[12];
          char   ftn_chars[12];
       }  comm_rec = { 1, 0, 1.23, 0.0, 2,4,6,8,10, 0,0,0,0,0,
                       "From C" };


main() /* C - FORTRAN  export/import of structs */
{
  struct {  int length;
            char *start;
         }  str_desc1, str_desc2;

  short i;

  str_desc1.start = &comm_rec.c_chars;
  str_desc1.length = strlen(comm_rec.c_chars);
  str_desc2.start = &comm_rec.ftn_chars;
  str_desc2.length = strlen(comm_rec.ftn_chars);

  ftn_sub();

  printf("\n\nC-main\n");
  printf("FORTRAN-integer    : ");
  printf("%2d \n", comm_rec.ftn_int);
  printf("FORTRAN-float      : ");
  printf("%8.2f \n", comm_rec.ftn_float);
  printf("FORTRAN-integer    : ");
  for (i = 0; i < 5; i++)
    printf("%2d ", comm_rec.ftn_iarr[i]);
  printf("\n");
  printf("FORTRAN-char-array : ");
  printf("%s \n", comm_rec.ftn_chars);
}
```

```
C     FP-STRUCT-1
      subroutine ftn_sub

      COMMON /comm_rec/c_int,ftn_int,c_float,ftn_float,
     *c_iarr,ftn_iarr,c_chars,ftn_chars

      INTEGER*4 c_int, ftn_int
      REAL*8 c_float, ftn_float
      INTEGER*4 c_iarr(0:4), ftn_iarr(0:4)
      CHARACTER*12 c_chars, ftn_chars
      INTEGER*2 I
C
100   FORMAT (X,'C-integer       : ',I2)
110   FORMAT (X,'C-double        : ',F8.2)
120   FORMAT (X,'C-integer-array : ',5(I2,x))
130   FORMAT (X,'C-char-array    : ',A12)
      WRITE (1,*) 'FORTRAN-subroutine'
      WRITE (1,100) c_int
      WRITE (1,110) c_float
      WRITE (1,120) c_iarr
      WRITE (1,130) c_chars
C
      ftn_int = 2
      ftn_float = 19.68
      DO 200 I=0,4
        FTN_iarr(i) = i * i
200   CONTINUE
      ftn_chars = 'From FORTRAN'
      RETURN
      END
```

Mode file to generate a C / FORTRAN program on ND-500

```
@delete-file,ftn-sub:nrf
@FORTRAN-500-K
compile ftn-sub:ftn,,"ftn-sub:nrf"
EXIT
@del-fil c-prog:NRF,,
@NC
option ic+
compile c-prog:C,,"c-prog:NRF"
EXIT
@LINKAGE-LOADER
abort-batch-on-error off
release-domain c-prog
delete-domain c-prog
abort-batch-on-error on
set-domain "c-prog"
open-segment "c-prog",,,
load c-prog
total-segment-load ftn-sub
local-trap-disable all
load nc-lib
load cat-lib
CC If the FORTRAN library is not defined as sharable
CC segment:
CC load fortran-lib
CC If the FORTRAN library is a sharable segment but
CC not an auto-link-segment:
CC link-segment (domain-user)fortran-lib
CC If the FORTRAN library is defined as
CC auto-link-segment, the segment will be linked
CC automatically with the exit command in the
CC linkage-loader
exit
```

Interfacing C and PLANC

main	It is recommended that C is the main program while PLANC is the subroutine.
compiler version	The PLANC routines were compiled with the F version of the PLANC-500 compiler.
hardware trap handlers	PLANC routines will not work properly with the hardware trap handlers enabled. So in the C program you have to disable the trap handlers every time you call a PLANC routine by means of clte(trap_nr) and to enable them after returning from the PLANC routine by means of sete(trap_nr).
stack	With routine standard you have to initialise a stack in the PLANC routine by means of the INISTACK routine. You must not use a stack with a non-standard routine.
invalues	You can only call PLANC routines that do not use invalues or outvalues.
export/import	When exporting/importing variables from/to PLANC the variable being initialised in C has to be imported. The PLANC routine and the variable initialised in PLANC have to be exported.
parameter list	If variables are exchanged via parameter list, only the routine has to be exported from PLANC.
	In general parameters are exchanged by reference i.e. the address operator has to be specified in the C program. Exceptions are mentioned above the example programs.
routine standard	With routine standard pointers to variables are used implicitly.
non standard routines	With non-standard routines the parameter list must contain the word "pointer" explicitly.

Export / import of integer variables

```
int c_int;
extern int planc_int;

main() /* C - PLANC export/import of integer */
{

  c_int = 1;
  clte(9);        /* switch off overflow trap handler */
  planc_sub();
  sete(9);        /* switch on overflow trap handler */
  printf("\n\nC-main\n");
  printf("PLANC-integer : %d\n", planc_int);
}

% PlncC-Int-1
MODULE planc_test
  EXPORT planc_sub, planc_int
  IMPORT Integer : c_int
  INTEGER : planc_int
  INTEGER ARRAY : stackarray(0:100)
  ROUTINE STANDARD void, void : planc_sub
    INISTACK stackarray
    OUTPUT (1,'A17','PLANC-Subroutine$')
    OUTPUT (1,'A12','C-integer : ')
    OUTPUT (1,'I2', c_int)
    4 =: planc_int
  ENDROUTINE
ENDMODULE
```

Integer variables as parameters (standard)

```
main() /* C - PLANC integer as parameter */
{
  int c_int = 1;
  int planc_int;

  clte(9);       /* switch off overflow trap handler */
  planc_sub(&c_int, &planc_int);
  sete(9);       /* switch on overflow trap handler */
  printf("\n\nC-main\n");
  printf("PLANC-integer : %2d\n", planc_int);
}


% Planc-C-Int-2
MODULE planc_test
  EXPORT  planc_sub
  INTEGER ARRAY : stackarray(0:100)
  ROUTINE STANDARD void, void (integer, integer read write) : &
                                                planc_sub(c_int, planc_int)
    INISTACK stackarray
    OUTPUT (1,'A17','PLANC-Subroutine$')
    OUTPUT (1,'A12','C-integer : ')
    OUTPUT (1,'I2', c_int)
  4 =: planc_int
ENDROUTINE
ENDMODULE
```

Integer variables as parameters (non standard)

```
main() /* C - PLANC integer as parameter */
      /* non standard routine          */
{
  int c_int;
  int planc_int;

  c_int = 1;
  clte(9);                /* switch off trap handlers */
  planc_sub(&c_int, &planc_int);
  sete(9);                 /* switch on trap handlers */
  printf("\n\nC-main\n");
  printf("PLANC-integer : %d\n", planc_int);
}
```

```
% Planc-C-Int-3
MODULE planc_test
  EXPORT  planc_sub
  ROUTINE void, void (integer pointer, integer pointer) : &
                                        planc_sub(c_int, planc_int)
    OUTPUT (1,'A17','PLANC-Subroutine$')
    OUTPUT (1,'A12','C-integer : ')
    OUTPUT (1,'I2', IND(c_int))
  4 =: IND(planc_int)
  ENDROUTINE
ENDMODULE
```

Export / import of real variables

```
double c_real;
extern double p_real;

main() /* C - PLANC  real as common */
{
  c_real = 1.23;
  clte(9); clte(13); clte(14);     /* switch off trap handlers */
  planc_sub();
  sete(9); sete(13); sete(14);     /* switch on trap handlers */
  printf("\n\nC-main\n");
  printf("PLANC-real : %8.2f\n", p_real);
}


% PlncC-real-1
MODULE planc_test
  EXPORT planc_sub, p_real
  IMPORT Real8 : c_real
  Real8 : p_real
  INTEGER ARRAY : stackarray(0:100)
  ROUTINE STANDARD void, void : planc_sub
    INISTACK stackarray
    OUTPUT (1,'A17','PLANC-Subroutine$')
    OUTPUT (1,'A9','C-real : ')
    OUTPUT (1,'F8.2', c_real)
    9.80 =: p_real
  ENDROUTINE
ENDMODULE
```

Real variables as parameters (standard)

```
main() /* C - PLANC  real variables as parameter */
       /* routine standard                       */
{
  double c_real, p_real;

  c_real = 1.23;
  clte(9); clte(13); clte(14);     /* switch off trap handlers */
  planc_sub(&c_real, &p_real);
  sete(9); sete(13); sete(14);     /* switch on trap handlers */
  printf("\n\nC-main\n");
  printf("PLANC-real : %4.2f\n", p_real);
}


MODULE planc_test
EXPORT planc_sub
INTEGER ARRAY : stackarray(0:100)
ROUTINE STANDARD void, void (real8, real8 read write) &
                                         : planc_sub(c_real, p_real);
    INISTACK stackarray
    OUTPUT (1,'A17','PLANC-Subroutine$')
    OUTPUT (1,'A9','C-real : ')
    OUTPUT (1,'F4.2', c_real)
    9.80 =: p_real
  ENDROUTINE
ENDMODULE
```

— Real variables as parameters (non standard)

```
main() /* C - PLANC  real variables as parameters */
      /* non standard routine                    */
{
  float c_real, p_real;

  c_real = 1.23;
  clte(9); clte(13); clte(14);     /* switch off trap handlers */
  planc_sub(&c_real, &p_real);
  sete(9); sete(13); sete(14);     /* switch on trap handlers */
  printf("\n\nC-main\n");
  printf("PLANC-real : %4.2f\n", p_real);
}


% PlncC-Real-3

MODULE planc_test
EXPORT planc_sub
ROUTINE void, void (real8 pointer, real8 pointer) &
                                        : planc_sub(c_real, p_real);
    OUTPUT (1,'A17','PLANC-Subroutine$')
    OUTPUT (1,'A9','C-real : ')
    OUTPUT (1,'F4.2', IND(c_real))
    9.80 =: IND(p_real)
  ENDROUTINE
ENDMODULE
```

— Export / import of integer arrays

```
int c_ints[5];
extern int planc_ints[5];

main()  /* C - PLANC  integer array  as common */
        /* routine standard                    */
{
  int  i;

  for (i = 0; i < 5; i++)
    c_ints[i] = 2 * i;
  clte(9); clte(25);
  planc_sub();
  sete(9); sete(25);
  printf("\n\nC-main \n");
  printf("PLANC-integer-array : ");
  for (i = 0; i < 5; i++)
    printf("%2d ",planc_ints[i]);
  printf("\n");
}


% PlncC-IntArr-1
MODULE PLANC_test
  EXPORT PLANC_sub, planc_ints
  TYPE int_arr = Integer ARRAY
  IMPORT int_arr : c_ints(0:4)
  INTEGER ARRAY : planc_ints(0:4)
  INTEGER ARRAY : stackarray(0:100)

    ROUTINE standard void,void : planc_sub
      INISTACK stackarray
      Integer : i
      OUTPUT (1,'A17','PLANC-Subroutine$')
      OUTPUT (1,'A18','C-integer-array : ')
      FOR i IN 0:4 DO
        OUTPUT (1,'I2', c_ints(i))
        OUTPUT (1,'A1',' ')
      ENDFOR
        FOR i IN 0:4 DO
      i =: planc_ints(i)
    ENDFOR
  ENDROUTINE
ENDMODULE
```

— Integer arrays as parameters (standard)

With routine standard integer arrays are exchanged by means of a descriptor which contains the address and the lower and upper bound of the array, which determine the number of elements. The descriptor has to be passed by reference.

```
main()  /* C - PLANC  integer array as parameter */
{
  struct {
    int *arrptr;
    int lb;       /* position of first array element */
    int ub;       /* position of last array element */
  } c_intarr_desc, planc_intarr_desc;
  int c_ints[5];
  int planc_ints[5];
  int i;

  for (i = 0; i < 5; i++)
    c_ints[i] = 2 * i;
  c_intarr_desc.arrptr = &c_ints;
  c_intarr_desc.lb = 0;
  c_intarr_desc.ub = 4;
  planc_intarr_desc.arrptr = &planc_ints;
  planc_intarr_desc.lb = 0;
  planc_intarr_desc.ub = 4;
  planc_sub(c_intarr_desc, planc_intarr_desc);
  printf("\n\nC-main\n");
  printf("PLANC-integer-array : ");
  for (i = 0; i < 5; i++)
    printf("%2d ", planc_ints[i]);
  printf("\n");
}

% PlncPas-IntArr-3 for ND500
MODULE PLANC_test
EXPORT  PLANC_sub
TYPE int_arr = Integer ARRAY
  INTEGER ARRAY : stackarray(0:100)
  ROUTINE standard void,void (int_arr pointer,int_arr pointer): &
                                                  PLANC_sub(c_ints,planc_ints)
    INISTACK stackarray
    Integer : i
    OUTPUT (1,'A17','PLANC-Subroutine$')
    OUTPUT (1,'A18','C-integer-array : ')
    FOR i IN 0:4 DO
      OUTPUT (1,'I2',IND(c_ints)(i))
      OUTPUT (1,'A1',' ')
    ENDFOR
    FOR i IN 0:4 DO
      i =: IND(planc_ints)(i)
    ENDFOR
  ENDROUTINE
ENDMODULE
```

Integer arrays as parameters (non standard)

```
main()  /* C - PLANC  integer arrays as parameters */
        /* non standard routine                    */
{
  int c_ints[5];
  int planc_ints[5];
  int i;

  for (i = 0; i < 5; i++)
    c_ints[i] = 2 * i;
  clte(9);
  planc_sub(&c_ints, 0, 4, &planc_ints, 0, 4);
  sete(9);
  printf("\n\nC-main\n");
  printf("PLANC-integer-array : ");
  for (i = 0; i < 5; i++)
    printf("%2d ", planc_ints[i]);
  printf("\n");
}


% PlncC-IntArr-2 for ND500
MODULE PLANC_test
EXPORT PLANC_sub
TYPE int_arr = Integer ARRAY

  ROUTINE void,void (int_arr pointer,int_arr pointer): &
                                        PLANC_sub(C_ints,PLANC_ints)
    Integer : i
    OUTPUT (1,'A17','PLANC-Subroutine$')
    OUTPUT (1,'A17','C-integer-array : ')
    FOR i IN 0:4 DO
      OUTPUT (1,'I2',IND(C_ints)(i))
      OUTPUT (1,'A1',' ')
    ENDFOR
    FOR i IN 0:4 DO
      i =: IND(PLANC_ints)(i)
    ENDFOR
  ENDROUTINE
ENDMODULE
```

Export / import of char arrays

```
char c_chars[7] = "From C";
extern char p_chars[10];

main()  /* C - PLANC  character array as common */
        /* routine standard                     */
{
  clte(9); clte(25);
  planc_sub();
  sete(9); sete(25);
  printf("\n\nC-main\n");
  printf("PLANC-char-array : ");
  printf("%s \n", p_chars);
}


% PlncPas-Char-1
MODULE PLANC_test
EXPORT  planc_sub, p_chars
TYPE char_arr = BYTES
IMPORT char_arr : c_chars(0:5)
bytes : p_chars(0:9)
INTEGER ARRAY : stackarray(0:100)

  ROUTINE STANDARD void, void : PLANC_sub
    INISTACK stackarray
    Integer : i
    OUTPUT (1,'A17','PLANC-Subroutine$')
    OUTPUT (1,'A15','C-char-array : ')
    OUTPUT (1,'A6', c_chars)
    'From PLANC' =: p_chars;
  ENDROUTINE
ENDMODULE
```

Char arrays as parameters (standard)

As the descriptors contain the addresses of the arrays address operator must not be specified in the parameter list.

```
main()  /* C - PLANC  char arrays as parameter */
        /* routine standard */
{
  struct {
           char *char_ptr;
           int lb;
           int ub;
         } c_char_desc, p_char_desc;

  char c_chars[7] = "From C";
  char p_chars[11];

  c_char_desc.char_ptr = &c_chars;
  c_char_desc.lb       = 0;
  c_char_desc.ub       = 5;
  p_char_desc.char_ptr = &p_chars;
  p_char_desc.lb       = 0;
  p_char_desc.ub       = 9;

  p_chars[10] = '\0';
  clte(9); clte(25);
  planc_sub(c_char_desc, p_char_desc);
  sete(9); sete(25);
  printf("\n\nC-main\n");
  printf("PLANC-char-array : ");
  printf("%s\n\n", p_chars);
}
```

```
% CPlnc-Char-3
MODULE PLANC_test
EXPORT  planc_sub
TYPE char_arr = BYTES
INTEGER array : stackarray(0:100)

  ROUTINE standard void, void (char_arr pointer, char_arr pointer) :&
                             planc_sub(c_chars, p_chars)
    INISTACK stackarray
    OUTPUT (1,'A17','PLANC-Subroutine$')
    OUTPUT (1,'A15','C-char-array : ')
    OUTPUT (1,'A11',IND(c_chars))
    'From PLANC ' =: IND(p_chars)
  ENDROUTINE
ENDMODULE
```

— Char arrays as parameters (non standard)

The address operator must not be specified for the parameter being initialised in PLANC.

```
main()  /* C - PLANC  char arrays as parameter */
        /* non standard routine                */

{
  char c_chars[7] = "From C";
  char p_chars[11];

  p_chars[10] = '\0';
  clte(9); clte(25);
  planc_sub(&c_chars, 0, 5, p_chars, 0, 9);  /* lower and upper */
                                              /* bounds of array */
  sete(9); sete(25);
  printf("\n\nC-main\n");
  printf("PLANC-char-array : ");
  printf("%s\n", p_chars);
}


% PlncPas-Char-2
MODULE PLANC_test
EXPORT  planc_sub
TYPE char_arr = BYTES

  ROUTINE void, void (char_arr pointer, char_arr pointer) : &
                                       PLANC_sub(c_chars, p_chars)
    OUTPUT (1,'A17','PLANC-Subroutine$')
    OUTPUT (1,'A15','C-characters : ')
    OUTPUT (1,'A6', IND(c_chars))
    'From PLANC' =: IND(p_chars)
  ENDROUTINE
ENDMODULE
```

Mode file to generate a C / PLANC program

```
@delete-file,ppp:nrf
@PLANC-500
compile ppp:plnc,,"ppp:nrf"
EXIT
@del-fil ccc:NRF,,
@NC
compile ccc:C,,"ccc:NRF"
EXIT
@LINK-LOAD
abort-batch-on-error off
rel-domain ccc
del-domain ccc
abort-batch-on-error on
set-domain "ccc"
open-seg "ccc",,,
local-trap-disable all
load ccc
total-segment-load ppp
load nc-lib
load cat-lib-b
load planc-lib-g00
list-entries-undefined
exit
@ND ccc
```

Interfacing C and PASCAL

compiler version

All PASCAL subroutines were compiled with the version B06 of the PASCAL-compiler.

libraries

The following routines are contained in the NC library as well as in the CAT library: cos, sin, cosh, sinh, exp, sqrt, time, index(string). If you use one of the routines in a PASCAL module and in a C module in the same program, without precaution the LINKAGE-LOADER would satisfy the references in a wrong way.

Solution: first load the C modules and the NC library. Then kill the names of the routine which are used in PASCAL and C modules with the KILL-ENTRIES command. After that you can load the PASCAL modules and the CAT library.

common blocks

In the C program the variables have to be declared above the main function. The variable being initialised in PASCAL must be declared as extern.

In the PASCAL module the variable initialised in C has to be imported. The module and the variable being initialised in PASCAL have to be exported.

parameter

The variable being initialised in C has to passed by value, the one being initialised in PASCAL by reference (address operator '&' in C, VAR in PASCAL). The module has to be exported from PASCAL.

Export / import of integer variables

```
#include <stdio.h>

extern int pas_int;
int c_int;

main()
{
  c_int = 1;
  pas_sub();
  printf ("\nC-main\n");
  printf ("PASCAL-integer : %d\n",pas_int);
}


MODULE PasC_Int_1(input, output);

EXPORTS pas_sub, pas_int;
IMPORTS c_int;

VAR
  PAS_int : Integer;
  C_int   : Integer;

PROCEDURE pas_sub;
BEGIN
  ReWrite(output, 0, 1);
  WriteLn(output, 'PASCAL-module');
  WriteLn(output, 'C-integer : ', c_int:2);
  pas_int := 2;
  Reset(output, 2);        { close output }
END;

BEGIN
END.
```

— Integer variables as parameters

```
#include <stdio.h>

main()
{
  int pas_int;
  int c_int;

  c_int = 1;
  pas_sub(c_int, &pas_int);
  printf ("\nC-main\n");
  printf ("PASCAL-integer : %d\n",pas_int);
}


MODULE PasC_Int_2(input, output);

EXPORTS pas_sub;

VAR
  PAS_int : Integer;
  C_int   : Integer;

PROCEDURE pas_sub(c_int : integer; VAR pas_int : integer);
BEGIN
  ReWrite(output, 0, 1);
  WriteLn(output, 'PASCAL-module');
  WriteLn(output, 'C-integer : ', c_int:2);
  pas_int := 2;
  Reset(output, 2);        { close output }
END;

BEGIN
END.
```

— Export / import of real variables

```
#include <stdio.h>

extern double pas_real;
double c_real;

main()
{
  c_real = 1.23;
  pas_sub():
  printf ("\nC-main\n");
  printf ("PASCAL-real : %6.2f\n",pas_real);
}


MODULE PasC_Real_1(input, output);

EXPORTS pas_sub, pas_real;
IMPORTS  c_real;

VAR
  PAS_real : real;
  C_real   : real;

PROCEDURE pas_sub;
BEGIN
  ReWrite(output, 0, 1);
  WriteLn(output, 'PASCAL-module');
  WriteLn(output, 'C-real : ', c_real:6:2);
  pas_real := 2.98;
  Reset(output, 2);        { close output }
END;

BEGIN
END.
```

Real variables as parameters

```
/* CPas-real-2 */
#include <stdio.h>


main()
{
  double pas_real;
  double c_real;

  c_real = 1.23;
  pas_sub(c_real, &pas_real);
  printf ("\nC-main\n");
  printf ("PASCAL-real : %6.2f\n",pas_real);
}


MODULE PasC_real_2(input, output);

EXPORTS pas_sub;

VAR
  PAS_real : real;
  C_real   : real;

PROCEDURE pas_sub(c_real : real; VAR pas_real : real);
BEGIN
  ReWrite(output, 0, 1);
  WriteLn(output, 'PASCAL-module');
  WriteLn(output, 'C-real : ', c_real:6:2);
  pas_real := 2.98;
  Reset(output, 2);        { close output }
END;

BEGIN
END.
```

Export / import of char arrays

When exchanging arrays of char with PASCAL the length of the array has to be passed explicitly.

```
/* CPas-char-1 */
#include <stdio.h>

extern  char pas_char[20];
char c_char[] = "From C";
int len;

main()
{
  len = strlen(c_char);
  pas_sub();
  printf ("\nC-main\n");
  printf ("PASCAL-char : %s\n",pas_char);
}


MODULE PasC_char_1(input, output);

EXPORTS pas_sub, pas_char;
IMPORTS c_char, len;

VAR
  PAS_char : packed array[0..19] of char;
  C_char   : packed array[0..19] of char;
  len      : integer;

PROCEDURE pas_sub;
BEGIN
  ReWrite(output, 0, 1);
  WriteLn(output, 'PASCAL-module');
  WriteLn(output, 'C-char : ', c_char:len);
  pas_char := 'From PASCAL'%00%;
  Reset(output, 2);        { close output }
END;

BEGIN
END.
```

Char arrays as parameters

As above the length of the array has to be passed explicitly.

```
#include <stdio.h>

main()
{
  char pas_char[20];
  char c_char[]= "From C";
  int len;

  len = strlen(c_char);
  pas_sub(c_char, len, &pas_char);
  printf ("\nC-main\n");
  printf ("PASCAL-char : %s\n",pas_char);
}


MODULE PasC_char_2(input, output);

EXPORTS pas_sub, char_arr;
TYPE
  char_arr = packed array[0..19] of char;
VAR
  PAS_char : char_arr;
  C_char   : char_arr;

PROCEDURE pas_sub(c_char : char_arr; len : Integer;
                  VAR pas_char : char_arr);
BEGIN
  ReWrite(output, 0, 1);
  WriteLn(output, 'PASCAL-module');
  WriteLn(output, 'C-char : ', c_char:len);
  pas_char := 'From PASCAL'%00%;
  Reset(output, 2);        { close output }
END;

BEGIN
END.
```

Export / import of structs

```
/* C - PASCAL struct 1 */
#include <stdio.h>
#include <string.h>

extern struct
{
  int c_int;
  int pas_int;
  double c_real;
  double pas_real;
  int c_ints[5];
  int pas_ints[5];
  char c_char[19];
  char pas_char[19];
  int len;
} common_record;

main()
{
  int i;
  common_record.c_int = 1;
  common_record.c_real = 1.23;
  for (i = 0; i < 5; i++)
    common_record.c_ints[i] = i;
  strcpy(common_record.c_char, "From C");
  common_record.len = strlen(common_record.c_char);
  pas_sub();
  printf ("\nC-main\n");
  printf ("PASCAL-integer       : %6d\n", common_record.pas_int);
  printf ("PASCAL-real          : %6.2f\n", common_record.pas_real);
  printf ("PASCAL-integer-array : ");
  for (i = 0; i < 5; i++)
    printf ("%2d ", common_record.pas_ints[i]);
  printf("\n");
  printf ("PASCAL-char          : %s\n",common_record.pas_char);
}
```

```
MODULE PasC_rec_1(input, output);

EXPORTS pas_sub;
IMPORTS common_record;

VAR
  common_record : RECORD
                    c_int    : integer;
                    pas_int  : integer;
                    c_real   : real;
                    pas_real : real;
                    c_ints   : array[0..4] of integer;
                    pas_ints : array[0..4] of integer;
                    c_char   : packed array[0..18] of char;
                    pas_char : packed array[0..18] of char;
                    len      : integer;
                  END;

PROCEDURE pas_sub;
VAR
  i : integer;
BEGIN
  ReWrite(output, 0, 1);
  WriteLn(output, 'PASCAL-module');
  WriteLn(output, 'C-integer       : ', common_record.c_int:6);
  WriteLn(output, 'C-real          : ', common_record.c_real:6:2);
  Write(output,   'C-integer-array : ');
  FOR i := 0 TO 4 DO
    Write(output, common_record.c_ints[i]:2,' ');
  WriteLn(output);
  WriteLn(output, 'C-char          : ',
                 common_record.c_char:common_record.len);
  common_record.pas_int := 2;
  common_record.pas_real := 2.98;
  FOR i := 0 TO 4 DO
    common_record.pas_ints[i] := i * 2;
  common_record.pas_char := 'From PASCAL'%00%;
  Reset(output, 2);        { close output }
END;

BEGIN
END.
```

Structs as parameters

As the struct contains the variables being used in both modules, only the struct has to be passed.

```
#include <stdio.h>
#include <string.h>
main()
{
  extern struct
  {
    int c_int;
    int pas_int;
    double c_real;
    double pas_real;
    int c_ints[5];
    int pas_ints[5];
    char c_char[19];
    char pas_char[19];
    int len;
  } common_record;
  int i;

  common_record.c_int = 1;
  common_record.c_real = 1.23;
  for (i = 0; i < 5; i++)
    common_record.c_ints[i] = i;
  strcpy(common_record.c_char, "From C");
  common_record.len = strlen(common_record.c_char);

  pas_sub(&common_record);

  printf ("\nC-main\n");
  printf ("PASCAL-integer       : %6d\n", common_record.pas_int);
  printf ("PASCAL-real          : %6.2f\n", common_record.pas_real);
  printf ("PASCAL-integer-array : ");
  for (i = 0; i < 5; i++)
    printf ("%2d ", common_record.pas_ints[i]);
  printf("\n");
  printf ("PASCAL-char          : %s\n",common_record.pas_char);
}
```

```
MODULE PasC_rec_2(input, output);

EXPORTS pas_sub, common_record;

TYPE
  common_record = RECORD
                    c_int    : integer;
                    pas_int  : integer;
                    c_real   : real;
                    pas_real : real;
                    c_ints   : array[0..4] of integer;
                    pas_ints : array[0..4] of integer;
                    c_char   : packed array[0..18] of char;
                    pas_char : packed array[0..18] of char;
                    len      : integer;
                  END;

PROCEDURE pas_sub(VAR com_rec : common_record);
VAR
  i : integer;
BEGIN
  ReWrite(output, 0, 1);
  WriteLn(output, 'PASCAL-module');
  WriteLn(output, 'C-integer        : ', com_rec.c_int:6);
  WriteLn(output, 'C-real           : ', com_rec.c_real:6:2);
  Write(output,   'C-integer-array  : ');
  FOR i := 0 TO 4 DO
    Write(output, com_rec.c_ints[i]:2,' ');
  WriteLn(output);
  WriteLn(output, 'C-char           : ', com_rec.c_char:com_rec.len);
  com_rec.pas_int := 2;
  com_rec.pas_real := 2.98;
  FOR i := 0 TO 4 DO
    com_rec.pas_ints[i] := i * 2;
  com_rec.pas_char := 'From PASCAL'%00%;
  Reset(output, 2);        { close output }
END;

BEGIN
END.
```

— Mode file to generate a C / PASCAL program —

```
@delete-file,ppp:nrf
@PASCAL
compile ppp:pasc,,"ppp:nrf"
EXIT
@del-fil ccc:NRF,,
@NC
CC option a1 for structs/records only
option a1
compile ccc:C,,"ccc:NRF"
EXIT
@LINK-LOAD
abort-batch-on-error off
rel-domain ccc
del-domain ccc
abort-batch-on-error on
set-domain "ccc"
open-seg "ccc",,,
local-trap-disable all
load ccc
total-segment-load ppp
load nc-lib
load cat-lib-b
list-entries-undefined
exit
@ND ccc
```

Chapter 15

Interfaces to the ND Environment

General

The C library contains definitions of interfaces to the standard subsystems SINTRAN III monitor calls, ISAM, SIBAS and FOCUS.

The functions of these packages can be called from your program without any external declaration. You only have to load the appropriate library in addition to the C and CAT libraries when linking your program. Details on how to load these libraries can be found in the appropriate manuals.

> **Note**
> ISAM-ND, SIBAS II and the FOCUS Screen Handling System are separate products; they do not belong to the C compiler package.

Monitor Call Interface

System functions of the SINTRAN operating system are called monitor calls. Detailed descriptions of the monitor calls can be found in the SINTRAN III Reference Manual (ND-60.128) and the SINTRAN III Monitor Calls Manual (ND-60.288).

The monitor call interface table starting on page 15-5 lists

- the function name which you have to use in your program,
- the name and number under which the monitor call is known in the SINTRAN operating system,
- the parameters you have to specify, and
- whether the monitor call sets the SINTRAN error code or not.

any type — Wherever *any type* is specified as parameter type, you have to pass the address of a variable, the type of which can be found in the description of the appropriate monitor call.

string — Wherever *string* is specified as parameter type three parameters have to be passed: the address of the string (&string), the starting position (0) and the length of the string (strlen(string)-1).

error code

Monitor calls that set the SINTRAN error code are marked with an asterisk in the column *ERR COD*. To check the result of a monitor call you can call the function *errcode* which has to be declared in your program as:

```
short errcode();
```

This function returns the SINTRAN error code. It should be called immediately after a monitor call. The value 0 (zero) indicates a successful execution of the latest call. If a Monitor call is not implemented -2 is returned.

example

```
main()
{
  short value;
  char s[] = "cc test";
  short echotable[16];
  short errcode();
  int error;

  /* monitor call 3: */
  SetEcho (1, 0, echotable, sizeof(echotable));

  /* monitor call 1: */
  InByte (1, &value);
  error = errcode();
  if (error != 0)
     printf ("SINTRAN error %d in InByte\n", error);

  /* monitor call 12: */
  SetCommandBuffer (s, 0, strlen(s)-1);
  ...
}
```

MONITOR CALL INTERFACE						
FUNCTION NAME	MONITOR CALL		PARAMETER		ERR COD	function
	NAME	NR	NAME	TYPE		
AccessRTCommon	RWRTC	406	1. func 2. rtcommon_addr 3. bytes 4. &buffer 5. sizeof(buffer)	long long long any type		Reads from or writes to RT common from an ND-500 program.
AdjustClock	CLADJ	112	1. number 2. time-unit	short short		Sets the computer's clock forward/back.
AppendSpooling	APSPF	240	1. &file_name 2. 0 3. strlen(file_name)-1 4. &spool_filename 5. 0 6. strlen(spool_filename)-1 7. copies 8. &user_text 9. 0 10. strlen(user_text)-1	string string short string	•	Prints a file by appending the file to the printer's output queue.
AssignCamacLam	ASSIG	154	1. ldn 2. lam 3. crate	short short short	•	Assigns a graded LAM to a logical device number.
AwaitFileTransfer	WAITF	121	1. fileno 2. ret_flag 3. &status	short short short		Checks that a data transfer to or from a mass storage file is completed.
AwaitRequest	WRQI	163	1. channel	short		Places the calling program in a waiting state.
AwaitTransfer	MWAITF	431	1. fileno 2. &return_flag 3. &bytes_read	long long long		Checks that a data transfer to or from a mass storage file is completed.
BackupClose	BCLOS	252	1. fileno 2. flag	short short	•	Closes a file.
BatchModeEcho	MBECH	325	1. control_bitmask	short		Controls echo of input and output in mode jobs.

MONITOR CALL INTERFACE						
FUNCTION NAME	MONITOR CALL		PARAMETER		ERR COD	function
	NAME	NR	NAME	TYPE		
BCNAF1Camac	BCNAF1	415	1. func 2. address 3. &data 4. &status	long long long long		Special CAMAC monitor call for the ND-500.
BCNAFCamac	BCNAF	414	1. func 2. address 3. &data 4. &status	long long long long		Special CAMAC monitor call for the ND-500.
BytesInBuffer	ISIZE	66	1. fileno 2. &bytes	short short	•	Gets the current number of bytes in the input buffer.
CallCommand	COMND	70	1. &str 2. 0 3. strlen(str)-1	string		Executes a SINTRAN command from a program.
CamacFunction	CAMAC	147	1. &data 2. &status 3. crate 4. station 5. subadr 6. funct	short short short short short short		Operate the CAMAC, i.e. execute NAF.
CamacGlRegister	GL	150	1. &data 2. crate	short short		Read the CAMAC GL register or the last CAMAC id-number.
CamacIOInstruction	IOXN	153	1. &data 2. ioxcode	short short		Executes a single IOX instruction.
CheckMonCall	MOINF	312	1. mon_call 2. &mon_entry	short short	•	Checks, if a particular monitor call exists in your
ClearCapability	CAPCLE	424	1. log_segment_numt 2. segment_type	long long	•	Clears a capability, which describes each logical segment in a domain.
ClearInBuffer	CIBUF	13	1. ldn	short	•	Clears a device input buffer.
ClearOutBuffer	COBUF	14	1. ldn	short	•	Clears a device output buffer.
CloseFile	CLOSE	43	1. fileno	short	•	Closes one or more files.

MONITOR CALL INTERFACE

FUNCTION NAME	MONITOR CALL NAME	MONITOR CALL NR	PARAMETER NAME	PARAMETER TYPE	ERR COD	function
CloseSpoolingFile	SPCLO	40	1. fileno 2. &str 3. 0 4. strlen(str)-1 5. copies 6. printflag	short string short short	•	Appends an opened file to a spooling queue.
CopyCapability	CAPCOP	423	1. source_log_seg_num 2. source_type 3. dest_log_seg_num 4. dest_type 5. access 6. &return_log_seg_num	long long long long long long	•	Copies a capability for a segment and the segment itself.
CreateFile	CRALF	221	1. &file_name 2. 0 3. strlen(file_name)-1 4. start 5. pages	string long long	•	Creates a file.
DataTransfer	ABSTR	131	1. ldn 2. func 3. mem_address 4. block_address 5. number_blocks 6. &status	short short long short short short		Transfers data between physical memory and a mass storage device.
DefaultRemoteSystem	SRUSI	314	1. &remote_system_name 2. 0 3. strlen(remote_system _name)-1 4. &remote_user_ident 5. 0 6. strlen(remote_user _ident)-1 7. &remote_user_passw 8. 0 9. strlen(remote_user _passw)-1 10. &remote_proj_passw 11. 0 12. strlen(remote_proj _passw)-1	string string string string	•	Sets default values for COSMOS.
DefineTermName (SetTerminalName)	STRFI	275	1. &file_name 2. 0 3. strlen(file_name)-1	string	•	Defines the file name to be used for terminals.

MONITOR CALL INTERFACE						
FUNCTION NAME	MONITOR CALL		PARAMETER		ERR COD	function
	NAME	NR	NAME	TYPE		
DelayStart	SET	101	1. rtadr 2. number 3. time_unit	short short short		Starts an RT-program after a specified time.
DeleteFile	MDLFI	54	1. &name_file 2. 0 3. strlen(name_file)-1	string	•	Deletes a file.
DeletePage	DELPG	272	1. fileno 2. first_page 3. last_page 4. &nr_pages	short long long long	•	Deletes pages from a file.
DeviceControl	IOSET	141	1. ldn 2. io_flag 3. rtadr 4. control 5. &status	short short short short short		Sets control information for a character device, eg. a terminal or a printer.
DeviceFunction	MAGTP	144	1. funct 2. &buffer 3. sizeof(buffer) 4. ldn 5. para1 6. &para2	short any type short short short	•	Performs various operations on floppy disks, magnetic tapes, cassette tapes etc..
DirectOpen	DOPEN	220	1. &fileno 2. access 3. &name 4. 0 5. strlen(name)-1 6. &typ 7. 0 8. strlen(typ)-1	short short string string	•	Opens a file.
DisableRTStart	RTOFF	137	1. rtadr	short		Disables start of RT programs.
DisAssemble	DISASS	401	1. program_pointer 2. &return_string 3. 0 4. sizeof(return_string)-1 5. max_num_chars	long string long		Disassembles one machine instruction on the ND-500.

MONITOR CALL INTERFACE						
FUNCTION NAME	MONITOR CALL		PARAMETER		ERR COD	function
	NAME	NR	NAME	TYPE		
DMAFunction	UDMA	333	1. ldn 2. func_code 3. &buffer 4. sizeof(buffer) 5. inp_para 6. &out_para	short short any type long long	*	Performs various DMA functions.
EnableRTStart	RTON	136	1. rtadr	short		Enables RT programs to be started.
EnterSegment	ENTSG	157	1. segment 2. pagetable 3. interruptlevel 4. startadr	short short short short		Enters a routine as a direct task or as a device driver.
ErrorMessage	QERMS	65	1. error	short		Displays a file system error message
EscapeDisable	DESCF	71	1. ldn	short		Disables the 'ESC'-key on the terminal.
EscapeEnable	EESCF	72	1. ldn	short		Enables the 'ESC'-key on the terminal.
ExactDelayStart	DSET	126	1. rtadr 2. basicunits	short long		Sets an RT program to start after a given period.
ExactInterval	DINTV	130	1. rtadr 2. basicunits	short long		Prepares an RT program for periodic execution.
ExactStartup	DABST	127	1. rtadr 2. basicunits	short long		Starts an RT program at a specific time.
ExecuteCommand	UECOM	317	1. &command 2. 0 3. strlen(command)-1	string		Executes a SINTRAN III command.
ExecutionInfo	RSIO	143	1. &excmode 2. &indev 3. &outdev 4. &usindex	short short short short		Gets information about the execution of a program.
ExitRTProgram	RTEXT	134				Terminates the calling RT or background program.

MONITOR CALL INTERFACE						
FUNCTION NAME	MONITOR CALL		PARAMETER		ERR COD	function
	NAME	NR	NAME	TYPE		
ExpandFile	EXPFI	231	1. &file_name 2. 0 3. strlen(file_name)-1 4. pages	string long	•	Expands the file size.
FileAsSegment	FSCNT	412	1. fileno 2. log_segment 3. segment_type 4. &segment_num	long long long long	•	Connects a file as a segment to your domain.
FileNotAsSegment	FSDCNT	413	1. fileno 2. segment_num	long long		Disconnects a file as a segment in your domain.
FileSystemFunction	FSMTY	327	1. func_code 2. fileno	short short	•	Makes sure that an uncontrolled system stop does not leave the file system inconsistent.
FindErrorDevice (GetErrorDevice)	GERDV	254	1. &error_device 2. &rt_address	short short		Gets the logical device number of the error device.
FindFileIndexes (GetFileIndexes)	FOBJN	274	1. &file_name 2. 0 3. strlen(file_name)-1 4. &file_type 5. 0 6. strlen(file_type)-1 7. &dir_index 8. &user_index 9. &object_index 10. &next_obj_index	string string short short short short	•	Gets the directory index, the user index and the object index of a file.
FindUserName (GetUserName)	GUSNA	214	1. &user_name 2. 0 3. sizeof(user_name)-1 4. dir_index 5. user_index	string short short	•	Gets the name of the user executing the program.
FixContiguous	FIXC	160	1. segment 2. pagetable 3. &status	short short short		Places a segment in physical memory.
FixInMemory	FIXMEM	410	1. mem_type 2. first_address 3. length 4. &nd100_address	long long long long		Fixes a logical segment of your domain in physical memory.

MONITOR CALL INTERFACE

FUNCTION NAME	MONITOR CALL		PARAMETER		ERR COD	function
	NAME	NR	NAME	TYPE		
FixScattered	FIX	115	1. segment	short		Place a segment in physical memory.
FullFileName	DEABF	256	1. &abfname	string	•	Returns a complete file name from an abbreviated one.
			2. 0			
			3. strlen(abfname)-1			
			4. &ffname	string		
			5. 0			
			6. sizeof(ffname)-1			
			7. &dftype	string		
			8. 0			
			9. strlen(dftype)-1			
GetActiveSegment	GASGM	421	1. &buffer	any type		Gets the names of segments in your domain.
			2. sizeof(buffer)			
GetAllFileIndexes	GUIOI	217	1. fileno	short	•	Gets the directory index, the user index and the object index of a file.
			2. &dir_index	short		
			3. &user_index	short		
			4. &object_index	short		
GetBasicTime	TIME	11	1. &int_time	long		Gets the current internal time.
GetBytesInFile	RMAX	62	1. fileno	short	•	Gets the number of bytes in a file.
			2. &bytes	long		
GetCurrentTime	CLOCK	113	1. &buffer	any type		Gets the current time and date.
			2. sizeof(buffer)			
GetDefaultDir	FDFDI	250	1. &user_name	string	•	Gets the user's default directory.
			2. 0			
			3. strlen(user_name)-1			
			4. &dir_index	short		
			5. &user_index	short		
GetDeviceType	GDEVT	263	1. ldn	short	•	Gets the device type eg. terminal, floppy disk, mass storage file etc..
			2. flag	short		
			3. &devtype	short		
			4. &devattr	long		
GetDirEntry	GDIEN	244	1. dir_index	short	•	Get information about a directory.
			2. &dir_entry	any type		
			3. sizeof(dir_entry)			
			4. &flag	short		

MONITOR CALL INTERFACE						
FUNCTION NAME	MONITOR CALL		PARAMETER		ERR COD	function
	NAME	NR	NAME	TYPE		
GetDirNameIndex	FDINA	243	1. &dir_name 2. 0 3. strlen(dir_name)-1 4. &dir_index 5. &name_index	string short short	•	Get directory index and name index.
GetDirUserIndexes	MUIDI	213	1. &user_name 2. 0 3. strlen(user_name)-1 4. &dir_index 5. &user_index	string short short	•	Gets a directory index and a user index.
GetErrorMessage	GETXM	334	1. error_code 2. &error_text 3. 0 4. sizeof(error_text)-1	short string	•	Gets a file system error message text.
GetEscLocalChars	MGDAE	230	1. ldn 2. &discon_char 3. &escape_char	short short short		Tells you which key to use to terminate a connection in a COSMOS network.
GetFileName	MGFIL	273	1. dir_index 2. user_index 3. object_index 4. &file_name 5. 0 6. sizeof(file_name)-1	short short short string	•	Gets the name of a file.
GetInputFlags	RFLAG	402	1. val	long		Gets the values of the ND-100/ND-500 communication flags.
GetLastByte	LASTC	26	1. ldn 2. &lastchar	short short	•	Gets the last character typed on a terminal.
GetNameEntry	GNAEN	245	1. name_index 2. &name_entry 3. sizeof(name_entry)	short any type	•	Gets information about devices.
GetND500Param	5PAGET	437	1. &buffer 2. sizeof(buffer)	any type		Gets information about why the last ND-500 program terminated.

MONITOR CALL INTERFACE

FUNCTION NAME	MONITOR CALL NAME	MONITOR CALL NR	PARAMETER NAME	PARAMETER TYPE	ERR COD	function
GetObjectEntry	DROBJ	215	1. &buffer	any type	•	Gets information about a file.
			2. sizeof(buffer)			
			3. dir_index	short		
			4. user_index	short		
			5. object_index	short		
GetOpenFileInfo (OpenFileInfo)	FOPFN	257	1. &file_name	string	•	Gets information about an open file.
			2. 0			
			3. strlen(file_name)-1			
			4. &file_type	string		
			5. 0			
			6. strlen(file_type)-1			
			7. &fileno	short		
			8. &access	short		
			9. &devno	short		
GetOwnProcessInfo	GPRNAME	427	1. &process_name	string		Gets the name and number of your process in the ND-500.
			2. 0			
			3. sizeof(process_name)-1			
			4. &process_number	long		
GetOwnRTAddress	GETRT	30	1. &rtadr	short		Gets the address of the calling program's RT description.
GetProcessNo	GPRNUM	426	1. &process_name	string		Gets the number of a process in the ND-500.
			2. 0			
			3. strlen(process_name)-1			
			4. &process_number	long		
GetRTAddress	GRTDA	151	1. &rtname	string		Gets the address of an RT description.
			2. 0			
			3. strlen(rtname)-1			
			4. &rtadr	short		
GetRTDescr	RTDSC	27	1. rtadr	short		Reads an RT description.
			2. &descriptor	any tpye		
			3. sizeof(descriptor)			
			4. &connected	short		
GetRTName	GRTNA	152	1. rtadr	short		Gets the name of an RT program.
			2. &rtname	string		
			3. 0			
			4. sizeof(rtname)-1			
GetScratchSegment	GSWSP	422	1. size_in_bytes	long	•	Connects an empty data segment to your domain.
			2. log_segment_num	long		
			3. &return_log_seg_num	long		

MONITOR CALL INTERFACE						
FUNCTION NAME	MONITOR CALL NAME	MONITOR CALL NR	PARAMETER NAME	PARAMETER TYPE	ERR COD	function
GetSegmentNo	GSGNO	322	1. &segment_name 2. 0 3. strlen(segment_name)-1 4. &segment_num	string short	•	Gets the number of a segment in the ND-100.
GetSpoolingEntry	RSPQE	55	1. spool_devno 2. &buffer 3. sizeof(buffer)	short any type	•	Gets the next spooling queue entry.
GetStartByte	REABT	75	1. fileno 2. &start	short long	•	Gets the number of the next byte to access.
GetSystemInfo	CPUST	262	1. number 2. &buffer 3. sizeof(buffer)	short any type	•	Gets various system information.
GetTerminalType	MGTTY	16	1. ldn 2. &typ	short short	•	Gets the terminals mode.
GetTrapReason	GERRCOD	505	1. &error_code	long		Gets the error code from the swapper process.
GetUserParam	PAGET	57	1. &buffer 2. sizeof(buffer)	any type		Gets information about why the last program terminated.
GetUserRegisters	GRBLK	420	1. &buffer 2. sizeof(buffer)	any type		Gets the contents of the registers, if 'ESC' terminates an ND-500 program.
GraphicFunction	GRAPHIC	155	1. x_coordinate 2. y_coordinate 3. code 4. ldn 5. funct 6. &status	short short short short short short		Executes various functions on a graphic peripheral, eg. NORDCOM terminal pen plotter, Textronix display.
In4x2Bytes	B4INW	63	1. ldn 2. num_bytes_read 3. &bytes 4. 0 5. sizeof(bytes)-1	short short string		Reads 8 bytes from a word- or character-oriented device.

MONITOR CALL INTERFACE						
FUNCTION NAME	MONITOR CALL		PARAMETER		ERR COD	function
	NAME	NR	NAME	TYPE		
In8AndFlag	T8INB	310	1. ldn 2. &nr_bytes 3. &buffer 4. 0 5. sizeof(buffer)-1	short short string		Reads 8 bytes from a device. Applies to the defined echo and break setting.
In8Bytes	B8INB	23	1. ldn 2. &chars 3. &str 4. sizeof(str)	short short any type	•	Reads 8 bytes from a device. Does not apply the defined echo and break setting.
InBufferState	IBRSIZ	313	1. ldn 2. &nr_in_buffer 3. &nr_until_break	short short short	•	Gets information about an input buffer.
InByte	INBT	1	1. ldn 2. &return	short short	•	Reads one character from a character device.
InputString	DVINST	503	1. ldn 2. max_num_bytes 3. &num_bytes_returned 4. &buffer 5. sizeof(buffer) 6. break_strategy 7. echo_strategy 8. break_table_1 9. break_table_2 10. break_table_3 11. break_table_4 12. echo_table_1 13. echo_table_2 14. echo_table_3 15. echo_table_4	long long long any type long long long long long long long long long long		Reads a string from a device, eg. a terminal.
InString	INSTR	161	1. ldn 2. &str 3. sizeof(str) 4. bytes 5. terminator 6. &status	short char short short short		Reads a string of characters from a peripheral device.
InUpTo8Bytes	M8INB	21	1. ldn 2. &chars 3. &str 4. sizeof(str)	short short any type	•	Reads up to 8 bytes from a device, eg. a terminal.

MONITOR CALL INTERFACE						
FUNCTION NAME	MONITOR CALL		PARAMETER		ERR COD	function
	NAME	NR	NAME	TYPE		
IOInstruction	EXIOX	31	1. &ain 2. regadr 3. &aout	short short short		Executes an IOX machine instruction.
MaxPagesInMemory	MXPISG	417	1. log_segment_num 2. segment_type 3. number_pages	long long long		Sets the maximum number of pages a segment may have in physical memory at a time.
MemoryUnFix	UNFIXMEM	411	1. address	long		Releases a fixed segment in your domain from physical memory.
ND500TimeOut	5TMOUT	514	1. nr_units 2. time_units 3. &status	long long long		Suspend the execution of an ND-500 program for a given time.
NewFileVersion	CRALN	253	1. &file_name 2. 0 3. strlen(file_name)-1 4. first_page 5. pages	string long long	•	Creates new versions of a file.
NewUser	SUSCN	241	1. &user_name 2. 0 3. strlen(user_name)-1 4. user_psw 5. &proj_name 6. 0 7. strlen(proj_name)-1 8. &status	string short string short	•	Switches the user name you are logged under.
NoInterruptStart	DSCNT	107	1. rtadr	short		Removes the connection of an RT-program to interrupts from a device.
NoWaitSwitch	NOWT	36	1. ldn 2. ioflag 3. waitflag	short short short	•	Switches NoWait on and off.
OldUser	RUSCN	242	1. user_type	short	•	Switches back to the user name you were logged in under before NewUser.

MONITOR CALL INTERFACE						
FUNCTION NAME	MONITOR CALL		PARAMETER		ERR COD	function
	NAME	NR	NAME	TYPE		
OpenFile	OPEN	50	1. &fileno	short	•	Opens a file.
			2. access	short		
			3. &name	string		
			4. 0			
			5. strlen(name)-1			
			6. &typ	string		
			7. 0			
			8. strlen(typ)-1			
Out8Bytes	B8OUT	24	1. ldn	short	•	Writes 8 bytes to a character device, eg. a terminal.
			2. &str	any type		
			3. sizeof(str)			
OutBufferSpace	OSIZE	67	1. fileno	short	•	Gets the current number of bytes in the output buffer.
			2. &bytes	short		
OutByte	OUTBT	2	1. ldn	short	•	Writes one byte to a character device.
			2. outval	short		
OutMessage	MSG	32	1. &str	string		Writes a message to the user's terminal.
			2. 0			
			3. strlen(str) - 1			
OutNumber	IOUT	35	1. format	short		Writes a number to a user's terminal.
			2. number	short		
OutputString	DVOUTS	504	1. ldn	long		Writes a string to a device, eg. a terminal.
			2. num_bytes	long		
			3. &buffer	any type		
			4. sizeof(buffer)			
OutString	OUTSTR	162	1. ldn	short		Writes a string of characters to a peripheral file, eg. a terminal.
			2. &str	char		
			3. sizeof(str)			
			4. bytes	short		
			5. &status	short		
OutUpTo8Bytes	M8OUT	22	1. ldn	short	•	Writes up to 8 bytes to a device.
			2. &str	any type		
			3. sizeof(str)			
PrivRelease	PRLS	125	1. ldn	short		Releases a device reserved by another RT program.
			2. io_flag	short		

MONITOR CALL INTERFACE						
FUNCTION NAME	MONITOR CALL		PARAMETER		ERR COD	function
	NAME	NR	NAME	TYPE		
PrivReserve	PRSRV	124	1. ldn 2. io_flag 3. rtadr 4. &status	short short short short		Reserves a device for another RT program.
ReadDiskPage	RDPAG	270	1. dir_index 2. &buffer 3. sizeof(buffer) 4. page 5. nr_pages	short any type long short	•	Reads one or more directory pages.
ReadFromFile	RFILE	117	1. ldn 2. wait 3. &buffer 4. sizeof(buffer) 5. blocknumber 6. words	short short any type short long	•	Reads any number of bytes from a file.
ReadObjectEntry	ROBJE	41	1. fileno 2. &buffer 3. sizeof(buffer)	short any type	•	Gets information about an opened file
ReadSegmentEntry (GetSegmentEntry)	RSEGM	53	1. segment 2. &buffer 3. sizeof(buffer)	short any type		Gets information about a segment in the ND 100.
ReadUserEntry (GetUserEntry)	RUSER	44	1. &user 2. 0 3. strlen(user) - 1 4. &destination 5. sizeof(destination)	string any type	•	Gets information about a user.
RelDirectory	RLDIR	247	1. dir_index	short	•	Releases a directory
ReleaseResource	RELES	123	1. ldn 2. io_flag	short short		Releases a reserved device or file.
RenameFile	MRNFI	232	1. &old_filename 2. 0 3. strlen(old_filename)-1 4. &new_filename 5. 0 6. strlen(new_filename)-1	string string	•	Renames a file.
ResDirectory (ReserveDir)	REDIR	246	1. dir_index	short	•	Reserves a directory for special use.

MONITOR CALL INTERFACE						
FUNCTION NAME	MONITOR CALL		PARAMETER		ERR COD	function
	NAME	NR	NAME	TYPE		
ReservationInfo	WHDEV	140	1. ldn 2. io_flag 3. &rtadr	short short short		Checks that a device is not reserved.
ReserveResource	RESRV	122	1. ldn 2. io_flag 3. ret_flag 4. &fstatus	short short short short		Reserves a device or file for your program only.
SaveND500Segment	WSEGN	416	1. log_segment_num 2. first_log_page 3. last_log_page	long long long	•	Writes all modified pages of a segment back to the disk.
SaveSegment	WSEG	164	1. segment	short		Saves a segment in the ND 100.
ScratchOpen	SCROP	235	1. &fileno 2. access 3. &name 4. 0 5. strlen(name)-1 6. &typ 7. 0 8. strlen(typ)-1	short short string string 	•	Opens a file as a scratch file.
SetBlockSize	SETBS	76	1. fileno 2. size	short long	•	Sets the block size of an opened file.
SetBreak	BRKM	4	1. ldn 2. strategy 3. &table 4. sizeof(table) 5. charnumber	short short any type short		Sets the break characters for a terminal.
SetBytePointer (SetStartByte)	SETBT	74	1. fileno 2. start	short long	•	Sets the next byte to be read or written in an opened mass storage file.
SetClock	UPDAT	111	1. minute 2. hour 3. day 4. month 5. year	short short short short short		Gives new values to the computer's clock and calendar.
SetCommandBuffer	SETCM	12	1. &str 2. 0 3. strlen(str) - 1	string		Transfers a string to the command buffer.

MONITOR CALL INTERFACE						
FUNCTION NAME	MONITOR CALL		PARAMETER		ERR COD	function
	NAME	NR	NAME	TYPE		
SetDirEntry (WriteDirEntry)	WDIEN	311	1. dir_index 2. &dir_entry 3. sizeof(dir_entry)	short any type	•	Changes the information about a directory.
SetEcho	ECHOM	3	1. ldn 2. strategy 3. &table 4. sizeof(table)	short short any type		Modifies a terminals echo.
SetEscLocalChars	MSDAE	227	1. ldn 2. discon_char 3. escape_char	short short short		Defines the 'ESC' and 'LOCAL' keys.
SetFileAccess	SFACC	237	1. &file_name 2. 0 3. strlen(file_name)-1 4. &public_access 5. 0 6. strlen(public_access)-1 7. &friend_access 8. 0 9. strlen(friend_access)-1 10. &own_access 11. 0 12. strlen(own_access)-1	string string string string	•	Sets the access pro tection for a file.
SetIOArea	IOFIX	404	1. first_address 2. bytes	long long		Fixes an address in a domain in physical memory.
SetMaxBytes	SMAX	73	1. fileno 2. bytes	short long	•	Sets the number of bytes in an opened file.
SetND500Param	5PASET	436	1. &buffer 2. sizeof(buffer)	any type		Sets information about an ND 500 program.
SetObjectEntry	DWOBJ	216	1. &buffer 2. sizeof(buffer) 3. dir_index 4. user_index 5. object_index	any type short short short	•	Changes the description of a file.
SetOutputFlags	WFLAG	403	1. val	long		Sets the communication flags between ND 100 and ND 500.

MONITOR CALL INTERFACE						
FUNCTION NAME	MONITOR CALL		PARAMETER		ERR COD	function
	NAME	NR	NAME	TYPE		
SetPeripheralName	SPEFI	234	1. &file_name 2. 0 3. strlen(file_name)-1 4. ldn	string short	•	Defines a periphal file, eg. a printer.
SetPermanentOpen	SPERD	236	1. fileno	short	•	Sets a file permanently open.
SetProcessName	SPRNAME	425	1. &process_name 2. 0 3. strlen(process_name)-1	string		Defines a new name for your process.
SetProcessPriority	SPRIO	507	1. new_priority	long		Sets the priority for a process.
SetRemoteAccess	SRLMO	316	1. mode	short		Switches remote file access on and off.
SetRTPriority	PRIOR	110	1. rtadr 2. prio 3. &oldprio	short short short		Sets the priority of an RT program.
SetStartBlock	SETBL	77	1. fileno 2. block	short short	•	Sets the next block to be read / written in an open file.
SetTemporaryFile	STEFI	233	1. &file_name 2. 0 3. strlen(file_name)-1	string	•	Defines a file to store information temporarily.
SetTerminalType	MSTTY	17	1. ldn 2. typ	short short	•	Sets the type of a terminal.
SetUserParam	PASET	56	1. &buffer 2. sizeof(buffer)	any type		Sets information about a background program.
StartInterval (StartupInterval)	INTV	103	1. rtadr 2. number 3. time_unit	short short short		Prepares an RT program for periodic execution.
StartOnInterrupt	CONCT	106	1. rtadr 2. ldn	short short		Connects an RT program to interrupts from a device.
StartProcess	STARTPR	500	1. process_number	long	•	Starts a process.
StartRTProgram	RT	100	1. rtadr	short		Starts an RT program

• SINTRAN version J only

MONITOR CALL INTERFACE						
FUNCTION NAME	MONITOR CALL		PARAMETER		ERR COD	function
	NAME	NR	NAME	TYPE		
StartTime (StartupTime)	ABSET	102	1. rtadr 2. seconds 3. minutes 4. hours	short short short short		Starts an RT program at a specified time of the day.
StopProcess	STOPPR	501				Sets the current process in a wait state.
StopProgram (ExitFromProgram)	LEAVE	0				Terminates a program.
StopRTProgram	ABORT	105	1. rtadr	short		Stops an RT program.
SuspendProgram	HOLD	104	1. number 2. time_unit	short short		Suspends the execu tion of your program for a given time.
SwitchProcess	SWITCHP	502	1. process_number	long		Sets the current process in a wait state and restarts another process.
SwitchUserBreak	USTBRK	405	1. func 2. address	long long		Switches user de-fined escape hand-ling on and off.
TermLineInfo (TerminalLineInfo)	TREPP	332	1. func_code 2. ldn 3. &status	short short short	•	Gets information about a terminal line.
TermMode (TerminalMode)	TERMO	52	1. ldn 2. mode	short short	•	Selects various terminal functions.
TermNoWait (TerminalNoWait)	TNOWAI	307	1. ldn 2. flag 3. no_wait 4. &status	short short short short		Switches 'no wait' on and off.
TermStatus (TerminalStatus)	TERST	330	1. ldn 2. &buffer 3. sizeof(buffer)	short any type	•	Gets information about a terminal.
TimeOut•	TMOUT	267	1. nr_units 2. time_units 3. &status	short short short	•	Suspends the execu-tion of your program for a given time.
TimeUsed	TUSED	114	1. &basicunits	long		Gets CPU time you have used.

MONITOR CALL INTERFACE						
FUNCTION NAME	MONITOR CALL		PARAMETER		ERR COD	function
	NAME	NR	NAME	TYPE		
ToErrorDevice	ERMON	142	1. error 2. suberror	short short		Outputs user defined real time error.
TranslateAddress	ADR100	430	1. nd500_array 2. &nd100_phys_word _address	long long		Translates an ND 500 logical address to an ND 100 physical address.
UnFixSegment	UNFIX	116	1. segment	short		Releases a fixed segment.
WaitForRestart	RTWT	135				Sets the RT program in a waiting state.
WarningMessage	ERMSG	64	1. error	short		Outputs a file system error message
WriteDiskPage	WDPAG	271	1. dir_index 2. &buffer 3. sizeof(buffer) 4. page 5. nr_pages	short any type long short	•	Writes to one or more pages in a directory.
WriteToFile	WFILE	120	1. ldn 2. wait 3. &buffer 4. sizeof(buffer) 5. blocknumber 6. words	short short any type short long	•	Writes any number of bytes to a file.

ISAM Interface

You can access indexed sequential files by calling the appropriate functions of the ISAM library. Details about ISAM are described in the Indexed Sequential Access Method Reference Manual (ND-60.108).

> **Note**
> The ISAM library has to be loaded after the C library (see page 11-15).

The ISAM interface table starting on page 15-31 lists

- the function name which you have to use in your program (these names correspond to the ISAM names extended by a *P* as third letter),
- the parameters you have to specify, and
- a short function description.

any type

Whenever *any type* is specified as parameter type, you have to pass the address of a variable, the type of which can be found in the description of the appropriate ISAM function.

string

Wherever *string* is specified as parameter type, three parameters have to be passed: the address of the string (&string), the starting position (0) and the end position (1) (= strlen(string)-1) of the string. As in ISAM file_id and key_id have a length of two characters the third parameter is always '1'.

short_array

The parameter type *short_array* of the function *ISPINI* is an array of five short integers. You have to pass three parameters for it: the address of the array, the starting position and the number of elements.

error return

If no errors occurred the parameter *status* returns the string '00'. Otherwise, it contains the ISAM error code.

EXCEPTION: The function *ISPINI* returns the error code as an integer value.

The following example is related to the problem at the beginning if the ISAM manual. It demonstrates the use of ISAM calls in a C program.

```
/* Example of how to use ISAM with C */

#include <stdio.h>
#include <string.h>
extern char *memset();
char xrecord[36];
char name[21];          /* real length=20, the additional char to hold the terminating '\0' */
char tele[5];           /*  4 */
char car_no[13];        /* 12 */
char name_search[] = "A                    ";
char tele_search[] = "0000";
char car_search[]  = "000000000000";
char status[3] = "00";
short inistat, err_code;
short fct;

main()
{
 int hmask=0;
 if (declare()) {                                              /* declare an ISAM file */
  printf("\nError in ISAM declaration, %2d Status = ", inistat);
  exit(1);
 }
                        /* Open ISAM file "SUPER-SYSTEM:DATA". This file has to exist on */
                        /* the current user. Note the " " behind the parameter filename. */
 ispopf("UM",0,1,"SU",0,1,"SUPER-SYSTEM:DATA ",0,17,1, &status[0],0,1);
 if ((strncmp(&status[0], "00", 2)) && (strncmp(&status[0], "94", 2))) {
  printf("\nError in ISAM-OPEN! Status = %c%c\n", status[0], status[1]);
  isperr(&err_code);
  printf("Error code of file system = %d\n", err_code);
  exit(1);
 }
                                  /* call "mask()" as long as there's no error (!=-1)  */
                                  /* and the user doesn't terminate the program.       */
 while (((hmask=mask()) != -1) && (hmask != 9));;
 ispclf("SU", 0, 1, &status[0], 0, 1);                       /* close ISAM file and exit */
 if (strncmp(&status[0], "00", 2)) {
  printf("\nError in ISAM-CLOSE! Status = %c%c\n", status[0], status[1]);
  isperr(&err_code);
  printf("Error code of file system = %d\n", err_code);
 }
}

short num(str)                                   /* help function for building the key(s) */
char *str;
{
  short i;
  i = *str * 256;
  str++;
  i = i + *str;
  return(i);
}
```

```
int declare()
{
  short info[5];
  short fct;

  inistat = 0;
  ispini(1, &info, 0, sizeof(info)/2-1, &inistat);          /* start definition */
  if (inistat) return(-1);

                                                     /* declare new ISAM file: */
  info[0] = num("SU");                                 /* file identification */
  info[1] = 36;                                        /* length of record    */
  info[2] = 0;
  info[4] = 0;
  ispini(2, &info, 0, sizeof(info)/2-1, &inistat);             /* new ISAM file */
  if (inistat) return(-1);

                                                          /* declare 3 keys: */
  info[1] = num("NA");                                       /* key id       */
  info[2] = 0;                                               /* key position */
  info[3] = 20;                                              /* key length   */
  info[4] = 0;
  ispini(3, &info, 0, sizeof(info)/2-1, &inistat);                 /* new key */
  if (inistat) return(-1);

  info[1] = num("TE");
  info[2] = 20;
  info[3] = 4;
  ispini(3, &info, 0, sizeof(info)/2-1, &inistat);
  if (inistat) return(-1);

  info[1] = num("CA");
  info[2] = 24;
  info[3] = 12;
  info[4] = 0;
  ispini(3, &info, 0, sizeof(info)/2-1, &inistat);
  if (inistat) return(-1);

                                     /* switch on internal ISAM buffering. */
  ispini(4, &info, 0, sizeof(info)/2-1, &inistat);
  if (inistat) return(-1);
  return(0);
}  /* declare */

int newrec()                                          /* write new record(s) */
{
 int c;
 short i;
 printf("%c%c", 0x19, 0x00);                                 /* clear screen */
 printf("\n");
 printf("%15s Please type name, phone number and car reg. no.\n"," ");
 printf("%15s (End with <CARRIAGE-RETURN> for <name>)\n"," ");
```

```
do {
 memset(&name[0], ' ', 20); name[20] = '\0';
 memset(&tele[0], ' ', 4);  tele[4] = '\0';
 memset(&car_no[0], ' ', 12); car_no[12] = '\0';
 printf("\n%15s Name         : "," ");
 if ((c = getchar()) == '\n') break;
 ungetc(c, stdin);
 i = 0;
 while (((c=getchar()) != '\n') && (i < 20)) {
  name[i] = c; i++;
 }
 if (c != '\n') while ((c=getchar()) != '\n');;
 printf("%15s Phoneno.     : "," ");
 i = 0;
 while (((c=getchar()) != '\n') && (i < 4)) {
  tele[i] = c; i++;
 }
 if (c != '\n') while ((c=getchar()) != '\n');;
 printf("%15s Car reg. no.: "," ");
 i = 0;
 while (((c=getchar()) != '\n') && (i < 12)) {
  car_no[i] = c; i++;
 }
 if (c != '\n') while ((c=getchar()) != '\n');;

 for (i = 0; i < strlen(&name[0]); i++)          xrecord[i] = name[i];
 for (i = strlen(&name[0]); i < 20; i++)         xrecord[i] = ' ';
 for (i = 20; i < 20+strlen(&tele[0]); i++)      xrecord[i] = tele[i-20];
 for (i = 20 + strlen(&tele[0]); i < 24; i++)    xrecord[i] = ' ';
 for (i = 24; i < 24+strlen(&car_no[0]); i++)    xrecord[i] = car_no[i-24];
 for (i = 24 + strlen(&car_no[0]); i < 36; i++) xrecord[i] = ' ';
 status[0] = '0'; status[1] = '0';
 ispwrt("SU", 0, 1, &xrecord[0], 0, 35, &status[0], 0, 1);
 if (strncmp(&status[0], "00", 2)) {
  printf("\n%15s Error in ISAM-WRITE! Status = %c%c\n", " ", status[0], status[1]);
  isperr(&err_code);
  printf("Error code of file system = %d\n", err_code);
  return(-1);
 }
} while (name[0] != ' ');
return(0);
} /* newrec */

int liste(which)
{
 int i;
 char *pname = "NAME                !";
 char *ptel = "TEL !";
 char *pcar = "CAR REG. NO.!";
 if (strncmp(&status[0], "00", 2) == 0) {
  if (which==1) printf("\n%s%s%s\n", pname, ptel, pcar);
  else if (which == 2) printf("\n%s%s%s\n", ptel, pname, pcar);
  else if (which == 3) printf("\n%s%s%s\n", pcar, pname, ptel);
  printf("====================================\n");
```

```
  while (strncmp(&status[0], "00", 2) == 0) {
   isprnx("SU", 0, 1, &xrecord[0], 36, &status[0], 0, 1);
   if (strncmp(&status[0], "00", 2)) {
    status[0] = '0'; status[1] = '0';
    break;
   }
   for (i = 0; i < 20; i++) name[i] = xrecord[i];
   name[20] = '\0';
   for (i = 20; i < 24; i++) tele[i - 20] = xrecord[i];
   tele[4] = '\0';
   for (i = 24; i < 36; i++) car_no[i - 24] = xrecord[i];
   car_no[12] = '\0';
   if ((which == 1) && (strncmp(&name[0], &name_search[0], 20)))
    printf("%20s!%4s!%12s!\n", name, tele, car_no);
   else if ((which == 2) && (strncmp(&tele[0], &tele_search[0], 4)))
    printf("%4s!%20s!%12s!\n", tele, name, car_no);
   else if ((which == 3) && (strncmp(&car_no[0], &car_search[0], 12)))
    printf("%12s!%20s!%4s!\n", car_no, name, tele);
  }
 }
 else {
  printf("\nError in ISAM-START! Status = %c%c\n", status[0], status[1]);
  return(-1);
 }
 return(0);
}  /* liste */

int namelist()
{
 short i;
 fct = 3;                                /* find record with key greater than given value */
 printf("%c%c\n",0x19,0x00);                                         /* clear screen */
 memset(&xrecord[0],' ',36);                               /* fill xrecord[] with blanks */
                          /* If (and only if) the key is identical to the first bytes of   */
                          /* the record, you may use the isptrtl function for searching,   */
                          /* i.e. the search key is given as parameter to the function,    */
                          /* which copies the key to the beginning of the record.          */
 isptrtl("SU", 0, 1, fct, "NA", 0, 1, &name_search[0], 20, &xrecord[0], 36, &status[0], 0, 1);
 if (liste(1)) return(-1);
 return(0);
}  /* namelist */

int telelist()
{
 short i;
 printf("%c%c\n", 0x19, 0x00);                                       /* clear screen */
 fct = 3;
                          /* As the record doesn't start with the key for the phone number */
                          /* you have to copy the search key to the correct place  within  */
                          /* the record, and call the "normal" ISAM function isptrt.       */
 memset(&xrecord[0], ' ', 20);
 for (i = 20; i < 24; i++) xrecord[i] = tele_search[20 - i];
 memset(&xrecord[24], ' ', 12);
```

```
 isptrt("SU", 0, 1, fct, "TE", 0, 1, &xrecord[0], 36, 36, &status[0], 0, 1);
 if (liste(2)) return(-1);
 return(0);
} /* telelist */

int carlist()
{
 short i;
 printf("%c%c\n", 0x19, 0x00);                                       /* clear screen */
 fct = 3;
 memset(&xrecord[0], ' ', 24);
 for (i = 24; i < 36; i++) xrecord[i] = car_search[24 - i];
 isptrt("SU", 0, 1, fct, "CA", 0, 1, &xrecord[0], 36, 36, &status[0], 0, 1);
 if (liste(3)) return(-1);
 return(0);
} /* carlist */

int del()                                                       /* delete record(s) */
{
 int c;
 short i;
 printf("%c%c\n", 0x19, 0x00);                                       /* clear screen */
 printf("%15s Name(s) of people to be deleted from file \n"," ");
 printf("%15s (End with <CARRIAGE-RETURN> for <Name>) \n"," ");;
 do {
  memset(&name[0], ' ', 20); name[20] = '\0';
  printf("\n%15s Name        : ", " ");
  if ((c = getchar()) == '\n') break;
  ungetc(c, stdin);
  i = 0;
  while (((c = getchar()) != '\n') && (i < 20)) {
   name[i] = c; i++;
  }
  if (c != '\n') while ((c = getchar()) != '\n');;
  for (i = 0; i < strlen(&name[0]); i++)  xrecord[i] = name[i];
  for (i = strlen(&name[0]); i < 36; i++) xrecord[i] = ' ';
  status[0] = '0'; status[1] = '0';
  ispdlk("SU", 0, 1, &xrecord[0], 36, 0, &status[0], 0, 1);
  if (strncmp(&status[0], "00", 2) == 0) printf("\n%15s Deleted!\n"," ");
  else {
   if ((strncmp(&status[0], "23", 2) == 0) && (name[0] != '\0'))
    printf("%15s Cannot find that name!\n"," ");
   else if (strncmp(&status[0], "23", 2)) {
    printf("%15s Error in ISAM-file! Status = %c%c\n"," ", status[0], status[1]);
    return(-1);
   }
  }
 } while (name[0] != ' ');
 return(0);
}  /* del */
```

```
int mask()
{
  int i, choice;
  char x;

  i = 0; choice = 0;
  printf("%c%c\n",0x19,0x00);                                             /* clear screen */
  printf("%30s M E N U\n\n"," ");
  printf("%15s Create new records                          :  1\n"," ");
  printf("%15s Listing, sorted according to names          :  2\n"," ");
  printf("%15s Listing, sorted according to phonenumbers   :  3\n"," ");
  printf("%15s Listing, sorted according to car reg. no.   :  4\n"," ");
  printf("%15s Delete records from file                    :  5\n"," ");
  printf("%15s Program end                                 :  9\n\n"," ");
  printf("%15s                              Input number :  "," ");
  choice = getchar(); if (choice == '\n') ungetc(choice, stdin);
  while ((x=getchar()) != '\n');;
  switch (choice)
  {
    case '1' : i=newrec();   break;
    case '2' : i=namelist(); break;
    case '3' : i=telelist(); break;
    case '4' : i=carlist();  break;
    case '5' : i=del();      break;
    case '9' : i=9;          break;
    default : printf("\n%15s Invalid number!"," ");
  } /* case */
  if ((i != -1) && (i!=9)) {
    printf("\n\n%15s Back to the menu with <CARRIAGE-RETURN> ! "," ");
    while ((x = getchar()) != '\n'); ;
  }
  return(i);
} /* mask */
```

I S A M I N T E R F A C E			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
ISPCLF	1. &file_id 2. 0 3. 1 4. &status 5. 0 6. 1	string string	Closes the file (ISAM-file process finished)
ISPCUR	1. &file_id 2. 0 3. 1 4. &record_no 5. &status 6. 0 7. 1	string long string	Returns the internal record number of the current record.
ISPDEL	1. &file_id 2. 0 3. 1 4. &status 5. 0 6. 1	string string	Deletes current record from the file.
ISPDIR	1. &file_id 2. 0 3. 1 4. direction 5. &status 6. 0 7. 1	string short string	Determines the read direction for an ISPRNX-call. Direction = 1 means in ascending sequence, direction = 2 in descending sequence.
ISPDLK	1. &file_id 2. 0 3. 1 4. &data_rec 5. sizeof(data_rec) 6. unused 7. &status 8. 0 9. 1	string any type string	Gets specified record and deletes it.
ISPERR	1. &err_code	short	Returns the error code of the file system.
ISPFLG	1. &file_id 2. 0 3. 1 4. &status 5. 0 6. 1	string string	Resets ISAM error flag.

ISAM INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
ISPFLU	1. &file_id 2. 0 3. 1 4. &status 5. 0 6. 1	string string	Flushes out buffered data.
ISPINF	1. &file_id 2. 0 3. 1 4. fct 5. &length 6. &number_deleted 7. &status 8. 0 9. 1	string short short long string	Gets information about an ISAM file.
ISPINI	1. fct 2. &info 3. 0 4. **sizeof**(info)/2-1 5. &status	short short_ array short	Initialises ISAM-buffer.
ISPLCK	-	-	Locks next record for a read or write call.
ISPOPF	1. &access 2. 0 3. strlen(access)-1 4. &file_id 5. 0 6. 1 7. &file_name 8. 0 9. strlen(file_name)-1 10. runmode 11. &status 12. 0 13. 1	string string string short string	Establishes link between file id (must be known to ISAM) and current file name and opens the file. The parameter *runmode* may have the values 0-3 (see ISOPF in the ISAM Reference Manual - ND-60.108).
ISPREL	1. &file_id 2. 0 3. 1 4. record_no 5. &data_rec 6. **sizeof**(data_rec) 7. &length 8. &status 9. 0 10. 1	string long any type short string	Reads a record directly by means of its internal record number. ISREL is considerably faster than ISPRNX or ISPRUK1

I S A M I N T E R F A C E			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
ISPREM	1. &file_id 2. 0 3. 1 4. &status 5. 0 6. 1	string string	Stores the current, internal record number.
ISPRES	1. &file_id 2. 0 3. 1 4. &status 5. 0 6. 1	string string	Resumes record access at the point marked by ISPREM.
ISPREW	1. &file_id 2. 0 3. 1 4. &data_rec 5. **sizeof**(data_rec) 6. &status 7. 0 8. 1	string any type string	Rewrites a record for files with constant record length.
ISPRGN	1. &file_id 2. 0 3. 1 4. &status 5. 0 6. 1	string string	Regenerates the index part from the data part.
ISPRKV1	1. &file_id 2. 0 3. 1 4. &key_id 5. 0 6. 1 7. &key_rec 8. **sizeof**(key_rec) 9. &data_rec 10. **sizeof**(data_rec) 11. &length 12. &status 13. 0 14. 1	string string any type any type short string	Reads record with specified key for files with variable record length.

ISAM INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
ISPRNV	1. &file_id	string	Reads sequentially the next record for files with variable record length (in byte).
	2. 0		
	3. 1		
	4. &data_rec	any type	
	5. **sizeof**(data_rec)		
	3. &length	**short**	
	6. &status	string	
	7. 0		
	8. 1		
ISPRNX	1. &file_id	string	Reads sequentially the next record for files with constant record length.
	2. 0		
	3. 1		
	4. &data_rec	any type	
	5. **sizeof**(data_rec)		
	6. &status	string	
	7. 0		
	8. 1		
ISPRUK1	1. &file_id	string	Reads record with specified key for files with constant record length.
	2. 0		
	3. 1		
	4. &key_id	string	
	5. 0		
	6. 1		
	7. &key_rec	any type	
	8. **sizeof**(key_rec)		
	9. &data_rec	any type	
	10. **sizeof**(data_rec)		
	11. &status	string	
	12. 0		
	13. 1		
ISPRWV	1. &file_id	string	Rewrites a record for files with variable record length.
	2. 0		
	3. 1		
	4. &data_rec	any type	
	5. **sizeof**(data_rec)		
	6. length	**short**	
	7. &status	string	
	8. 0		
	9. 1		

I S A M I N T E R F A C E			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
ISPTRT	1. &file_id	string	Gets a record by comparing a specified value to its key.
	2. 0		
	3. 1		
	4. fct	short	
	5. &key_id	string	
	6. 0		
	7. 1		
	10. &data_rec	any type	
	11. **sizeof**(data_rec)		
	12. unused		
	13. &status	string	
	14. 0		
	15. 1		
ISPTRT1	1. &file_id	string	Gets a record by comparing a specified value to its key.
	2. 0		
	3. 1		
	4. fct	short	
	5. &key_id	string	
	6. 0		
	7. 1		
	8. &key_rec	any type	
	9. **sizeof**(key_rec)		
	10. &data_rec	any type	
	11. **sizeof**(data_rec)		
	12. &status	string	
	13. 0		
	14. 1		
ISPUNL	1. &file_id	string	Unlocks all locked records of the file.
	2. 0		
	3. 1		
	4. &status	string	
	5. 0		
	6. 1		
ISPVER	1. &file_id	string	Checks consistency between index and data parts of an ISAM file.
	2. 0		
	3. 1		
	4. &status	string	
	5. 0		
	6. 1		

I S A M I N T E R F A C E			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
ISPWRT	1. &file_id 2. 0 3. 1 4. &data_rec 5. **sizeof**(data_rec) 6. &status 7. 0 8. 1	string any type string	Writes a record into the file and updates the index tables for files with constant record length.
ISPWRV	1. &file_id 2. 0 3. 1 4. &data_rec 5. **sizeof**(data_rec) 6. length 7. &status 8. 0 9. 1	string any type short string	Writes a record into the file and updates the index tables for files with variable record length.

SIBAS Interface

Details about SIBAS are described in the SIBAS II User Manual (ND-60.127).

> **Note**
> The SIBAS library has to be loaded after the C library (see page 11-15).

The SIBAS interface table starting on page 15-39 lists

- the function name which you have to use in your program (these names correspond to the SIBAS names extended by a *P* as first letter),
- the parameters you have to specify, and
- a short function description.

any type

Wherever *any type* is specified as parameter type, you have to pass the address of a variable, the type of which can be found in the description of the appropriate SIBAS function.

string

Wherever *string* is specified as parameter type three parameters have to be passed: the address of the string (&string), the starting position (0) and the length of the string (strlen(string)-1).

error return

If no errors occurred the parameter *status* returns the value 0. Otherwise, it contains the SIBAS error code.

example

```
main()
{
  short mode;
  short status;
  char dbname[] = "dbase";
  char password[] = "Fred";
  char realm_names = "realm1__realm2";
  int usage_mode[2], protection_mode[2];

  PSOPDB (mode, &dbname, 0, strlen(dbname)-1,
          &password, 0, strlen(password)-1,
          &status);

  if (status != 0) goto error;

  PSRRLM1 (2, &realm_names, 0, strlen(realm_names)-1,
           &usage_mode, sizeof(usage_mode)-1,
           &protection_mode,
            sizeof(protection_mode)-1,
           &status);

  if (status != 0) goto error;
  ...
  error:
  ...
  ...
}
```

S I B A S I N T E R F A C E			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
PACCDD1	1. RowID	**short**	accumulate double integer
	2. NoOfColumns	**short**	
	3. &ColumnList	string	
	4. 0		
	5. strlen(ColumnList)-1		
	6. &ValuesToAdd	any type	
	7. **sizeof**(ValuesToAdd)		
	8. &AccumValues	any type	
	9. **sizeof**(AccumValues)		
	10. &Status	**short**	
PACCFD1	1. RowID	**short**	accumulate floating
	2. NoOfColumns	**short**	
	3. &ColumnList	string	
	4. 0		
	5. strlen(ColumnList)-1		
	6. &ValuesToAdd	any type	
	7. **sizeof**(ValuesToAdd)		
	8. &AccumValues	any type	
	9. **sizeof**(AccumValues)		
	10. &Status	**short**	
PACCID1	1. RowID	**short**	accumulate integer
	2. NoOfColumns	**short**	
	3. &ColumnList	string	
	4. 0		
	5. strlen(ColumnList)-1		
	6. &ValuesToAdd	any type	
	7. **sizeof**(ValuesToAdd)		
	8. &AccumValues	any tpye	
	9. **sizeof**(AccumValues)		
	10. &Status	**short**	
PBSEQU	1. &TransActName	string	begin sequence
	2. 0		
	3. strlen(TransActName)-1		
	4. &Status	**short**	
PESEQU	1. &TransActName	string	end sequence
	2. 0		
	3. strlen(TransActName)-1		
	4. &Status	**short**	

SIBAS INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
PGMDFY1	1. &TableName	string	fetch and modify
	2. 0		
	3. strlen(TableName)-1		
	4. &IndexName	string	
	5. 0		
	6. strlen(IndexName)-1		
	7. &IndexValue	any type	
	8. **sizeof**(IndexValue)		
	9. IndexValueLength	**short**	
	10. NoOfColumns	**short**	
	11. &ColumnList	string	
	12. 0		
	13. strlen(ColumnList)-1		
	14. &Values	any type	
	15. **sizeof**(Values)		
	16. &Status	**short**	
	17. ValueLength	**short**	
PGRASE	1. &TableName	string	fetch and erase row
	2. 0		
	3. strlen(TableName)-1		
	4. &IndexName	string	
	5. 0		
	6. strlen(IndexName)-1		
	7. &IndexValue	any type	
	8. **sizeof**(IndexValue)		
	9. IndexValueLength	**short**	
	10. OptionCode	**short**	
	11. &Status	**short**	
PINLOG	1. &UserArea	string	initiate logging
	2. 0		
	3. strlen(UserArea)-1		
	4. &DatabaseName	string	
	5. 0		
	6. strlen(DatabaseName)-1		
	7. OptionCode	**short**	
	8. &Directory	string	
	9. 0		
	10. strlen(Directory)-1		
	11. LogInfo	**short**	
	12. NoOfPages	**short**	
	13. &Status	**short**	
POFLOG	1. &Status	**short**	set R-log off
PONLOG	1. &Status	**short**	set R-log on
PRELDV	1. ProcessNo	**short**	release process number
	2. &Status	**short**	

S I B A S I N T E R F A C E			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
PRELSI	1. &Status	short	release SIBAS
PRESIB	1. &Status	short	reserve SIBAS
PSABOR	1. &DatabaseName	string	abort user
	2. 0		
	3. strlen(DatabaseName)-1		
	4. &Status	short	
	5. UserID	short	
PSACT	1. &ProcessNo	short	activate SIBAS process
	2. &MachineName	string	
	3. 0		
	4. **sizeof**(MachineName)-1;		
	5. &DatabaseName	string	
	6. 0		
	7. strlen(DatabaseName)-1		
	8. &UserArea	string	
	9. 0		
	10. **sizeof**(UserArea)-1;		
	11. &DbAdminName	string	
	12. 0		
	13. **sizeof**(DbAdminName)-1		
	14. &DbAdminPassW	string	
	15. 0		
	16. strlen(DbAdminPassW)-1		
	17. &ProjektPassW	string	
	18. 0		
	19. strlen(ProjektPassW)-1		
	20. &Status	short	
	21. &ErrorInfo	short	
PSCHPO	1. &CheckpointId	any type	define checkpoint
	2. **sizeof**(CheckpointId)		
	3. &Status	short	
PSCHPW	1. &RowPassword	string	change password
	2. 0		
	3. strlen(RowPassword)-1		
	4. &Status	short	
PSCLDB	1. &DatabaseName	string	close database
	2. 0		
	3. strlen(DatabaseName)-1		
	4. &Status	short	

SIBAS INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
PSCONA	1. ConnectRowID 2. PositionID 3. &SetReferralName 4. 0 5. strlen(SetReferralName)-1 6. &Status	short short string short	connect after
PSCONB	1. ConnectRowId 2. PositionID 3. &SetReferralName 4. 0 5. strlen(SetReferralName)-1 6. &Status	short short string short	connect before
PSCONN	1. ConnectRowId 2. &SetReferralName 3. 0 4. strlen(SetReferralName)-1 5. &status	short string short	connect
PSDCON	1. ConnectRowID 2. &SetReferralName 3. 0 4. strlen(SetReferralName)-1 5. &Status	short string short	disconnect
PSDBEC	1. &LastSetRefName 2. 0 3. **sizeof**(LastSetRefName)-1 4. &LastTabName 5. 0 6. **sizeof**(LastTabName)-1 7. &ErrorTabName 8. 0 9. **sizeof**(ErrorTabName)-1 10. &LastColName 11. 0 12. **sizeof**(LastColName)-1 13. &DmlCallCode 14. &Dbec	string string string string short short	accept
PSEMSG	1. OptionCode 2. &InInfo 3. **sizeof**(InInfo) 4. &OutInfo 5. **sizeof**(OutInfo) 6. &Status	short any type any type short	reading SSI-SEC code and user and log info

SIBAS INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
PSEREL1	1. RowID	**short**	erase column values
	2. NoOfColumns	**short**	
	3. &ColumnList	**string**	
	4. 0		
	5. strlen(ColumnList)-1		
	6. &Status	**short**	
PSETDV	1. ProcessNo	**short**	set process number
PSEXMC	1. &InInfo	any type	execute user macro
	2. **sizeof**(InInfo)		
	3. InInfoLen	**short**	
	4. &OutInfo	any type	
	5. **sizeof**(OutInfo)		
	6. &OutInfoLen	**short**	
	7. &Status	**short**	
PSFEBL	1. &TableName	string	find first between limits
	2. 0		
	3. strlen(TableName)-1		
	4. &IndexName	string	
	5. 0		
	6. strlen(IndexName)-1		
	7. &LowLimit	any type	
	8. **sizeof**(LowLimit)		
	9. &HighLimit	any type	
	10. **sizeof**(HighLimit)		
	11. &Status	**short**	
	12. IndexValueLength	**short**	
PSFINI	1. &Status	**short**	finish recovery
PSFLBL	1. &TableName	string	find last between limits
	2. 0		
	3. strlen(TableName)-1		
	4. &IndexName	string	
	5. 0		
	6. strlen(IndexName)-1		
	7. &LowLimit	any type	
	8. **sizeof**(LowLimit)		
	9. &HighLimit	any type	
	10. **sizeof**(HighLimit)		
	11. &Status	**short**	
	12. IndexValueLength	**short**	
PSFORG	1. RowID/SearchRegionID	**short**	forget
	2. OptionCode	**short**	
	3. &Status	**short**	

S I B A S I N T E R F A C E			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
PSFRGT1	1. &TableName	string	find using row number + get
	2. 0		
	3. strlen(TableName)-1		
	4. PhysRowNo	long	
	5. NoOfColumns	short	
	6. &ColumnList	string	
	7. 0		
	8. strlen(ColumnList)-1		
	9. &Values	any type	
	10. **sizeof**(Values)		
	11. &Status	short	
PSFRLM1	1. NoOfTables	short	finish table
	2. &TableNames	string	
	3. 0		
	4. strlen(TabelNames)-1		
	5. &Status	short	
PSFRNO	1. &TableName	string	find using row number
	2. 0		
	3. strlen(TableName)-1		
	4. PhysRowNo	long	
	5. &Status	short	
PSFTCH	1. &TableName	string	direct find
	2. 0		- find using key
	3. strlen(TableName)-1		
	4. &IndexName	string	
	5. 0		
	6. strlen(IndexName)-1		
	7. &IndexValue	any type	
	8. **sizeof**(IndexValue)		
	9. &Status	short	
	10. IndexValueLength	short	

SIBAS INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
PSFTGT1	1. &TableName	string	fetch get
	2. 0		
	3. strlen(TableName)-1		
	4. &IndexName	string	
	5. 0		
	6. strlen(IndexName)-1		
	7. IndexValueLength	**short**	
	8. &IndexValue	any type	
	9. **sizeof**(IndexValue)		
	10. NoOfItems	**short**	
	11. &ItemList	string	
	12. 0		
	13. strlen(ItemList)-1		
	14. &Values	any type	
	15. **sizeof**(Values)		
	16. &Status	**short**	
PSGET1	1. RowID	**short**	get a row
	2. NoOfColumns	**short**	
	3. &ColumnList	string	
	4. 0		
	5. strlen(ColumnList)-1		
	6. &Values	any type	
	7. **sizeof**(Values)		
	8. &Status	**short**	
PSGETN1	1. RowID	**short**	get n rows
	2. SearchRegionID	**short**	
	3. NoWanted	**short**	
	4. NoOfColumns	**short**	
	5. &ColumnList	string	
	6. 0		
	7. strlen(ColumnList)-1		
	8. &Values	any type	
	9. **sizeof**(Values)		
	10. &NoFound	**short**	
	11. &Status	**short**	
PSGEWD	1. RowID	**short**	get wide
	2. &ColumnName	any type	
	3. **sizeof**(ColumnName)		
	4. OptionCode	**short**	
	5. ValueLengthWanted	long	
	6. &ControlBlock	any type	
	7. **sizeof**(ControlBlock)		
	8. &ValueLengthFound	long	
	9. &Values	any type	
	10. **sizeof**(Values)		
	11. &Status	**short**	

SIBAS INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
PSGIXN	1. RowID	**short**	get index values
	2. SearchRegionID	**short**	
	3. NoWanted	**short**	
	4. &Values	any type	
	5. **sizeof**(Values)		
	6. &NoFound	**short**	
	7. &Status	**short**	
PSICON	1. OptionCode	**short**	set reprocessing condition
	2. UserID	**short**	
	3. &Time	any type	
	4. **sizeof**(Time)		
	5. &TransActName	string	
	6. 0		
	7. strlen(TransActName)-1		
	8. &Status	**short**	
PSINFO	1. OptionCode	**short**	get schema information
	2. &Tab_Set_TextNam	string	
	3. 0		
	4. strlen(Tab_Set_TextNam)-1		
	5. &ColumnName	string	
	6. 0		
	7. strlen(ColumnName)-1		
	8. &OutInfoLength	**short**	
	9. &OutInfo	any type	
	10. **sizeof**(OutInfo)		
	11. &Status	**short**	
PSINSR	1. RowID	**short**	insert index value
	2. &IndexName	string	
	3. 0		
	4. strlen(IndexName)-1		
	5. &Status	**short**	
PSISTA	1. &OutInfo	any type	get SIBAS status
	2. **sizeof**(OutInfo)		
	3. &status	**short**	
PSLOCK	1. RowID	**short**	lock row
	2. OptionCode	**short**	
	3. &Status	**short**	

SIBAS INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
PSMDFY1	1. RowID	short	modify
	2. NoOfColumns	short	
	3. &ColumnList	string	
	4. 0		
	5. strlen(ColumnList)-1		
	6. &Values	any type	
	7. sizeof(Values)		
	8. &Status	short	
	9. ValueLength	short	
PSMESS	1. InInfoLength	short	message to R-log
	2. &InInfo	string	
	3. 0		
	4. strlenf(InInfo)-1		
	5. &Status	short	
PSOPDB	1. Usage	short	open database
	2. &DatabaseName	string	
	3. 0		
	4. strlen(DatabaseName)-1		
	5. &Password	string	
	6. 0		
	7. strlen(Password)-1		
	8. &Status	short	
PSPASS	1. &Status	short	set passive
PSPAUS	1. &Status	short	pause
PSRASE	1. RowID	short	erase row
	2. OptionCode	short	
	3. &Status	short	
PSRECO	1. &Status	short	recover
PSREMB	1. &SearchRegionID	short	remember
	2. OptionCode	short	
	3. &Status	short	
PSREMO	1. RowID	short	remove index value
	2. &IndexName	string	
	3. 0		
	4. strlen(IndexName)-1		
	5. &Status	short	

SIBAS INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
PSREPR	1. Condition 2. OptionCode 3. &Time 4. **sizeof**(Time) 5. NoOfCalls 6. PrintOption 7. UserID 8. RemoveFlag 9. &Status	short short any type short short short short short	reprocess R-log
PSRFIR	1. &TableName 2. 0 3. strlen(TableName)-1 4. &Status	string short	find first in table
PSRFSM	1. ReferenceRowID 2. &SetReferralName 3. 0 4. strlen(SetReferralName)-1 5. &Status	short string short	relative find - find first in set
PSRGET1	1. &RowID 2. &SearchRegionID 3. Direction 4. OptionCode 5. NoOfNames 6. &NameList 7. 0 8. strlen(NameList)-1 9. &PhysRowNo 10. &Values 11. **sizeof**(Values) 12. &Status	short short short short short string long any type short	find relatively + get/erase
PSRLSM	1. ReferenceRowID 2. &SetRefferalName 3. 0 4. strlen(SetReferralName)-1 5. &Status	short string short	find last in set
PSRNIS	1. RowID 2. SearchRegionID 3. &Status	short short short	find next in search region
PSRNSM	1. ReferringRowID 2. &SetReferralName 3. 0 4. strlen(SetReferralName)-1 5. &Status	short string short	find next in set

SIBAS INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
PSROLL	1. &CheckpointID	any type	rollback
	2. **sizeof**(CheckpointId)		
	3. &DatabaseName	string	
	4. 0		
	5. strlen(DatabaseName)-1		
	6. &DbPassword	string	
	7. 0		
	8. strlen(DbPassword)-1		
	9. &Status	short	
PSRPIS	1. RowID	short	find prior in search region
	2. SearchRowID	short	
	3. &Status	short	
PSRPSM	1. ReferringRowID	short	find prior in set
	2. &SetReferralName	string	
	3. 0		
	4. strlen(SetReferralName)-1		
	5. &Status	short	
PSRRLM1	1. NoOfTables	short	ready table
	2. &TableNames	string	
	3. 0		
	4. strlen(TableNames)-1		
	5. &Usages	any type	
	6. **sizeof**(Usages)		
	7. &Protections	short	
	8. **sizeof**(Protections)-1		
	9. &Status	short	
PSRSOW	1. ReferringRowID	short	find using foreign key
	2. &SetReferralName	string	
	3. 0		
	4. strlen(SetReferralName)-1		
	5. &status	short	
PSRUN	1. RunInfo	short	run SIBAS
	2. &Status	short	
PSSCUR	1. OptionCode	short	set current
	2. Row	long	
	3. SearchRegionID	short	
	4. &TableName	string	
	5. 0		
	6. strlen(TableName)-1		
	7. &Status	short	

SIBAS INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
PSSGET1	1. RowID	short	super get
	2. NoOfColumns	short	forget + remember + get
	3. &ColumnList	string	
	4. 0		
	5. strlen(ColumnList)-1		
	6. &Values	any type	
	7. **sizeof**(Values)		
	8. &Status	short	
PSSTA	1. OptionCode	short	get process status
	2. &ProcessNo	short	
	3. &ActiveState	short	
	4. &MachineName	string	
	5. 0		
	6. **sizeof**(MachineName)-1		
	7. &DatabaseName	string	
	8. 0		
	9. **sizeof**(DatabaseName)-1		
	10. &UserArea	string	
	11. 0		
	12. **sizeof**(UserArea)-1;		
	13. &DbAdminName	string	
	14. 0		
	15. **sizeof**(DbAdminName)-1		
	16. &Communication	short	
	17. &CPUType	short	
	18. &SIBASFiles	string	
	19. 0		
	20. **sizeof**(SIBASFiles)-1		
	21. &SIBASVersion	string	
	22. 0		
	23. **sizeof**(SIBASVersion)-1		
	24. &Remote	short	
	25. &Status	short	
	26. &ErrorInfo	short	
PSTART	1. &UserArea	string	start SIBAS
	2. 0		
	3. strlen(UserArea)-1		
	4. &DatabaseName	string	
	5. 0		
	6. strlen(DatabaseName)-1		
	7. Dummy	short	
	8. &Status	short	
PSTGET	1. &State	short	get process state
	2. &Status	short	
PSTOPS	1. &Status	short	stop SIBAS

S I B A S I N T E R F A C E			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
PSTORE1	1. &TableName	string	store
	2. 0		
	3. strlen(TableName)-1		
	4. NoOfColumns	short	
	5. &ColumnList	string	
	6. 0		
	7. strlen(ColumnList)-1		
	8. &Values	any type	
	9. **sizeof**(Values)		
	10. &Status	short	
	11. ValueLength	short	
PSTOWD	1. RowId	short	store wide
	2. &ColumnName	string	
	3. 0		
	4. strlen(ColumnName)-1		
	5. OptionCode	short	
	6. ValueLengthToStore	long	
	7. &ControlBlock	any type	
	8. **sizeof**(ControlBlock)		
	9. &Values	any type	
	10. **sizeof**(Values)		
	11. &Status	short	
PSTRLG	1. TerminalNo	short	terminal log
	2. OptionCode	short	
	3. &Status	short	
PSUBEG	1. UserID	short	transaction begin
	2. Dummy	short	
	3. &Status	short	
PSUEND	1. UserID	short	transaction end
	2. commit	short	
	3. &status	short	
PSUNLK	1. &Status	short	unlock row
PSUSIN	1. Priority	short	user information
	2. Dummy	short	
	3. Dummy	short	
	4. Dummy	short	
	5. &Status	short	
PSWHAT	1. RowID	short	what is row
	2. &TableName	string	
	3. 0		
	4. **sizeof**(TableName)-1		
	5. &PhysRowNo	long	
	6. &Status	short	

SIBAS INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
PSWINF	1. OptionCode	**short**	write schema information
	2. &Tab_Set_Txt_Nam	string	
	3. 0		
	4. strlen(Tab_Set_Txt_Nam)-1		
	5. &ColumnName	string	
	6. 0		
	7. strlen(ColumnName)-1		
	8. InInfoLength	**short**	
	9. &InInfo	any type	
	10. **sizeof**(InInfo)		
	11. &Status	**short**	
PSYNCP	1. OptionCode	**short**	synchronised checkpoint
	2. &Checkpoint	any type	
	3. **sizeof**(checkpoint)		
	4. &Status	**short**	
PUTBLK	1. &Status	**short**	write R-log block

FOCUS Interface

Details about FOCUS are described in the FOCUS Screen Handling System Reference Manual (ND-60.137).

> **Note**
> The FOCUS library has to be loaded after the C library (see page 11-15).

The FOCUS interface table starting on page 15-68 lists

- the function name which you have to use in your program (these names correspond to the FOCUS names with the second letter *C* replaced by a *P* or a *1*),
- the parameters you have to specify, and
- a short function description.

any type — Wherever *any type* is specified as parameter type, you have to pass the address of a variable, the type of which can be found in the description of the appropriate FOCUS function.

string — Wherever *string* is specified as parameter type three parameters have to be passed: the address of the string (&string), the starting position (0) and the length of the string (strlen(string)-1).

short_array — Wherever *short_array* is specified as parameter you have to consult the description of the appropriate FOCUS function to find out what is expected. You have to pass three parameters for it: the address of the array, the starting position and the number of elements.

struct alignment — In case you have trouble with structs passed to FOCUS calls you should try two byte or even one byte record alignment (option a1, a2 respectively) when compiling the C program.

attributes

For the attributes required in function *F1SETAT* you can define the following macros:

```
#define HIGH_INTENSITY 0x80000000
#define LOW_INTENSITY  0x40000000
#define ITALICS        0x20000000
#define UNDERLINED     0x10000000
#define SLOW_BLINK     0x08000000
#define RAPID_BLINK    0x04000000
#define INVERSE_VIDEO  0x02000000
#define INVISIBLE      0x01000000
```

The attribute(s) wanted have to be assigned to a four-byte integer variable. The address of the variable has to be passed to the FOCUS call. You can combine several attributes by using the bitwise OR operator (|). Please note that the character under the cursor looks different from the other characters in the specified field.

terminal type

The set of attributes which are available to you depend on the terminal type. The Tandberg TDV 2200 for instance doesn't support the italic attribute.

status_struct
status_record

The parameter type *status_struct* in the call *FPDESB* must be declared in your program as:

```
struct STATUS_RECORD {
       short term_code;
       short term_char;
       short fields_edited;
       short fields_in_edset;
       char  field_name[8];
       short occ_number; }
```

and a variable of type STATUS_RECORD has to be defined as global variable.

error return

If no errors occurred the parameter *status* returns the value 0. Otherwise, it contains the FOCUS error code.

example

The following program demonstrates the use of FOCUS screen attributes in C. The program assumes the that a FOCUS formfile exists, which contains the used forms. The formfile and the forms have to be created with FOCUS-DEF.

The following form is used by the example program.

```
--------------------------------------------------------------------------------

                                   F O R M

  Form name  : FORM1                              Size    : 508
```

```
  Form start : line 5   column 15
  Form end   : line 12  column 44                    Written : 1989-08-04
```

F O R M   L A Y O U T

```
          High intensity: ..............
          Low intensity : ..............
          Italics       : ..............
          Underlined    : ..............
          Slow blink    : ..............
          Rapid blink   : ..............
          Inverse Video : ..............
          Invisible     : ..............
```

F I E L D S

Name	Field format	LD	LS	Help	St/Ju	BWZ	Occ
FIELD1	X(14)	14	14		None	No	L
FIELD2	X(14)	14	14		None	No	L
FIELD3	X(14)	14	14		None	No	L
FIELD4	X(14)	14	14		None	No	L
FIELD5	X(14)	14	14		None	No	L
FIELD6	X(14)	14	14		None	No	L
FIELD7	X(14)	14	14		None	No	L
FIELD8	X(14)	14	14		None	No	L

F I E L D S   A N D   O C C U R R E N C E S

Line	Column	Field name
5	31	FIELD1
6	31	FIELD2
7	31	FIELD3
8	31	FIELD4
9	31	FIELD5
10	31	FIELD6
11	31	FIELD7
12	31	FIELD8

```
/* Small example of "set attributes" in FOCUS */

#include <stdio.h>
#define HIGH_INTENSITY 0x80000000
#define LOW_INTENSITY  0x40000000
#define ITALICS        0x20000000
#define UNDERLINED     0x10000000
#define SLOW_BLINK     0x08000000
#define RAPID_BLINK    0x04000000
#define INVERSE_VIDEO  0x02000000
#define INVISIBLE      0x01000000
int attr;
short foc_init_arr[10];                                    /* FOCUS initiation array */
char form_buffer[2048];                                        /* FOCUS form buffer */
short foc_stat = 0;                               /* status returned from FOCUS calls */

main()
{
 int i;
 char c;
 char *cp1 = "high_intensity";
 char *cp2 = "low_intensity ";
 char *cp3 = "italics       ";
 char *cp4 = "underlined    ";
 char *cp5 = "slow_blink    ";
 char *cp6 = "rapid_blink   ";
 char *cp7 = "inverse_video ";
 char *cp8 = "invisible     ";

 foc_init_arr[0] = 2048;                          /* length of form buffer in bytes */
 foc_init_arr[1] = 1;                               /* background terminal is used */
 foc_init_arr[2] = foc_init_arr[3] = 0;                                /* not used */
 foc_init_arr[4] = 1;                             /* word length: 1 word -> 1 byte */
 for (i = 5; i < 10; i++) foc_init_arr[i] = 0;                         /* not used */
 fpinite(&foc_init_arr[0], 0, strlen(&foc_init_arr[0])-1,
                                   &form_buffer, sizeof(form_buffer)-1, &foc_stat);
 if (foc_stat) {
  printf("Error in FOCUS initialisation.\n");
  exit(1);
 }
 fpdecff("C-FOCUS:FORM", 0, 11, &foc_stat);
 if (foc_stat) {
  fpgmsge("ERROR in declare form. Press <CR> to finish. '", 0, 29, &c, 0, 0, &foc_stat);
  foc_init_arr[1] = 0;
  fpinite(&foc_init_arr[0], 0, strlen(&foc_init_arr[0])-1,
                                   &form_buffer, sizeof(form_buffer)-1, &foc_stat);
  exit(1);
 }
```

```
fpdecfn("FORM1'", 0, 5, 0, &foc_stat);
if (foc_stat) {
 fpgmsge("ERROR! Press <CR> to finish. '", 0, 29, &c, 0, 0, &foc_stat);
 exit(1);
}
fldecrc("FIELD1'", 0, 6, 1, 1, &foc_stat);
attr = HIGH_INTENSITY;
flsetat(&attr,"FIELD1'", 0, 6, 1, 1, &foc_stat);
fpwfld("FIELD1'", 0, 6, 1, cp1, 13, 14, &foc_stat);

fldecrc("FIELD2'", 0, 6, 1, 1, &foc_stat);
attr = LOW_INTENSITY;
flsetat(&attr,"FIELD2'", 0, 6, 1, 1, &foc_stat);
fpwfld("FIELD2'",0, 6, 1, cp2, 13, 14, &foc_stat);

fldecrc("FIELD3'", 0, 6, 1, 1, &foc_stat);
attr = ITALICS;
flsetat(&attr,"FIELD3'", 0, 6, 1, 1, &foc_stat);
fpwfld("FIELD3'", 0, 6, 1, cp3, 13, 14, &foc_stat);

fldecrc("FIELD4'", 0, 6, 1, 1, &foc_stat);
attr = UNDERLINED;
flsetat(&attr,"FIELD4'", 0, 6, 1, 1, &foc_stat);
fpwfld("FIELD4'", 0, 6, 1, cp4, 13, 14, &foc_stat);

fldecrc("FIELD5'", 0, 6, 1, 1, &foc_stat);
attr = SLOW_BLINK;
flsetat(&attr, "FIELD5'", 0, 6, 1, 1, &foc_stat);
fpwfld("FIELD5'", 0, 6, 1, cp5, 13, 14, &foc_stat);

fldecrc("FIELD6'", 0, 6, 1, 1, &foc_stat);
attr = RAPID_BLINK;
flsetat(&attr, "FIELD6'",0, 6, 1, 1, &foc_stat);
fpwfld("FIELD6'", 0, 6, 1, cp6, 13, 14, &foc_stat);

fldecrc("FIELD7'", 0, 6, 1, 1, &foc_stat);
attr = INVERSE_VIDEO;
flsetat(&attr,"FIELD7'", 0, 6, 1, 1, &foc_stat);
fpwfld("FIELD7'", 0, 6, 1, cp7, 13, 14, &foc_stat);

fldecrc("FIELD8'", 0, 6, 1, 1, &foc_stat);
attr = INVISIBLE;
flsetat(&attr, "FIELD8'", 0, 6, 1, 1, &foc_stat);
fpwfld("FIELD8'", 0, 6, 1, cp8, 13, 14, &foc_stat);

fpgmsge("Press <CR> to finish. ", 0, 21, &c, 0, 0, &foc_stat);
foc_init_arr[1] = 0; /* to terminate FOCUS */
fpinite(&foc_init_arr[0], 0, sizeof(foc_init_arr)-1,
                                   &form_buffer, 0, sizeof(form_buffer)-1, &foc_stat);
if (foc_stat) {
 printf("\nError %4d in terminating FOCUS.\n", foc_stat);
}
}
```

The following program uses ISAM and FOCUS. The data handling is done in ISAM while the screen handling is done in FOCUS. The following form is used:

```
-----------------------------------------------------------------------------

                                   F O R M

  Form name  : MENU                                 Size    : 770

  Form start : line 3    column 13
  Form end   : line 16   column 58                  Written : 1989-08-01

-----------------------------------------------------------------------------

                            F O R M   L A Y O U T

=============================================================================

                         M A I N   M E N U

           +---------------------------------------------+
           | Create new records                       : 1 |
           | Listing, according to names              : 2 |
           | Listing, according to phone numbers      : 3 |
           | Listing, according to car reg. numbers   : 4 |
           | Delete records from ISAM file            : 5 |
           | Program end                              : 9 |
           +---------------------------------------------+
           | Please enter the number of                   |
           | the function you want to execute         : . |
           +---------------------------------------------+

=============================================================================

                               F I E L D S


Name       Field format                       LD  LS  Help      St/Ju  BWZ  Occ
--------   ---------------------------------  --  --  --------  -----  ---  ---
FUNCNO     X                                  1   1             None   No   L
    - Legal values   : '1','2','3','4','5','9'
```

```
F I E L D S   A N D   O C C U R R E N C E S
--------------------------------------------

Line   Column   Field name
----   ------   ----------
  15     56      FUNCNO

------------------------------------------------------------------------------
```

```
/* Example of how to use ISAM and FOCUS with C */

#include <stdio.h>
#include <string.h>
extern char *memset();
#define UNUSED 0
#define HIGH_INTENSITY 0x80000000
#define LOW_INTENSITY  0x40000000
#define ITALICS        0x20000000
#define UNDERLINED     0x10000000
#define SLOW_BLINK     0x08000000
#define RAPID_BLINK    0x04000000
#define INVERSE_VIDEO  0x02000000
#define INVISIBLE      0x01000000
int attr;
short foc_init_arr[10];                                     /* FOCUS initiation array */
char form_buffer[2048];                                          /* FOCUS form buffer */
short foc_stat = 0;                                /* status returned from FOCUS calls */
char err_buf[74] = "Error nnnn in xxxxx                         Press <CR> to finish '";
char *err_fc = "FOCUS";
char *err_is = " ISAM";
short ret_len;
char xrecord[38];
char name[24]; /* real length=23, the additional to hold the terminating '\0' */
char tele[4];     /*  3 */
char car_no[13]; /* 12 */
char name_search[] = "A                       ";
char tele_search[] = "000";
char car_search[]  = "000000000000";
char status[3] = "00";
short inistat, err_code;
short fct;

main()
{
 int i;
 int hmask=0;
 if (declare()) { /* declare an ISAM file */
  printf("\nError in ISAM declaration, %2d Status = ", inistat);
  exit(1);
 }
                             /* Open ISAM file "SUPER-SYSTEM:DATA". This file has to exist on the */
                             /* current user. Note the " " behind the parameter filename:        */
 ispopf("UM", 0, 1, "SU", 0, 1, "SUPER-SYSTEM:DATA ", 0, 17, 1, &status[0], 0, 1);
```

```
 if ((strncmp(&status[0], "00", 2)) && (strncmp(&status[0], "94", 2))) {
  printf("\nError in ISAM-OPEN! Status = %c%c\n", status[0], status[1]);
  isperr(&err_code);
  printf("Error code of file system = %d\n", err_code);
  exit(1);
 }
 foc_init_arr[0] = 2048;                          /* length of form buffer in bytes */
 foc_init_arr[1] = 1;                               /* background terminal is used */
 foc_init_arr[2] = foc_init_arr[3] = 0;                                 /* not used */
 foc_init_arr[4] = 1;                            /* word length: 1 word -> 1 byte */
 for (i=5; i<10; i++) foc_init_arr[i] = 0;                              /* not used */
 fpinite(&foc_init_arr[0], 0, strlen(&foc_init_arr[0])-1,
                                   &form_buffer, sizeof(form_buffer)-1, &foc_stat);
 if (foc_stat) {
  print_error(1, "initialisation.                 ");
  exit(1);
 }
 fpdecff("C-FOCUS:FORM", 0, 11, &foc_stat);
 if (foc_stat) {
  print_error(1, "form file declaration.          ");
  foc_init_arr[1] = 0;
  fpinite(&foc_init_arr[0], 0, strlen(&foc_init_arr[0])-1,
                                   &form_buffer, sizeof(form_buffer)-1, &foc_stat);
  exit(1);
 }
                              /* call "mask()" as long as there's no error (! = -1)  */
                              /* and the user doesn't terminate the program.         */
 while (((hmask=mask())!=-1) && (hmask!=9));;
 ispclf("SU", 0, 1, &status[0], 0, 1);                     /* close ISAM file and exit */
 if (strncmp(&status[0], "00", 2)) {
  print_error(0, "close.                          ");
 }
 foc_init_arr[1] = 0; /* to terminate FOCUS */
 fpinite(&foc_init_arr[0], 0, sizeof(foc_init_arr)-1,
                                &form_buffer, 0, sizeof(form_buffer)-1, &foc_stat);
 if (foc_stat) {
  printf("\nError %4d in terminating FOCUS.\n", foc_stat);
 }
}

short num(str)                             /* help function for building the key(s) */
char *str;
{
  short i;
  i = *str * 256;
  str++;
  i = i + *str;
  return(i);
}
```

```
void print_error(is_focus, err_str)
int is_focus;
char *err_str;
{
 char c;
 if (is_focus) sprintf(&err_buf[6], "%4d", foc_stat);
 else          sprintf(&err_buf[6], "  %c%c", status[0], status[1]);
 err_buf[10] = ' ';                    /* needed to overwrite the '\0' added by sprintf */
 if (is_focus) strncpy(&err_buf[14], err_fc, 5);
 else          strncpy(&err_buf[14], err_is, 5);
 strncpy(&err_buf[20], err_str, 32);
 fpgmsge(&err_buf[0], 0, 72, &c, 0, 0, &foc_stat);
}

int declare()
{
  short info[5];
  short fct;

  inistat = 0;
  ispini(1, &info, 0, sizeof(info)/2-1, &inistat);                   /* start definition */
  if (inistat) return(-1);

                                                                 /* declare new ISAM file: */
  info[0] = num("SU");                                             /* file identification */
  info[1] = 38;                                                       /* length of record */
  info[2] = 0;
  info[4] = 0;
  ispini(2, &info, 0, sizeof(info)/2-1, &inistat);                      /* new ISAM file */
  if (inistat) return(-1);

                                                                       /* declare 3 keys: */
  info[1] = num("NA");                                                        /* key id */
  info[2] = 0;                                                          /* key position */
  info[3] = 23;                                                           /* key length */
  info[4] = 0;
  ispini(3, &info, 0, sizeof(info)/2-1, &inistat);                             /* new key */
  if (inistat) return(-1);

  info[1] = num("TE");
  info[2] = 23;
  info[3] = 3;
  ispini(3, &info, 0, sizeof(info)/2-1, &inistat);
  if (inistat) return(-1);

  info[1] = num("CA");
  info[2] = 26;
  info[3] = 12;
  info[4] = 0;
  ispini(3, &info, 0, sizeof(info)/2-1, &inistat);
  if (inistat) return(-1);
                                                   /* switch on internal ISAM buffering: */
  ispini(4, &info, 0, sizeof(info)/2-1, &inistat);
  if (inistat) return(-1);
  return(0);
 }  /* declare */
```

```
int newrec()                                                  /* write new record(s) */
{
 int c;
 short i;
 short nam_len, tel_len, car_len;
 fpdecfn("NEW-REC", 0, 6, 0, &foc_stat);
 if (foc_stat) {
  print_error(1,"form name declaration(NEW-REC). ");
  return(-1);
 }
 fldecrc("NAME PHONE CARNO'", 0, 16, 1, 1, &foc_stat);
 if (foc_stat) {
  print_error(1,"record declaration.                    ");
  return(-1);
 }
 do
 {
  fpefld("NAME", 0, 3, 1, &name[0], 22, &nam_len, 1, &foc_stat);
  if (foc_stat) {
   print_error(1,"edit one field(name).           ");
   return(-1);
  }
  if (nam_len == 0) break;
  fpefld("PHONE", 0, 4, 1, &tele[0], 2, &tel_len, 1, &foc_stat);
  if (foc_stat) {
   print_error(1,"edit one field(phone).          ");
   return(-1);
  }
  fpefld("CARNO", 0, 4, 1, &car_no[0], 11, &car_len, 1, &foc_stat);
  if (foc_stat) {
   print_error(1, "edit one field(carno).          ");
   return(-1);
  }
  for (i = 0; i < nam_len; i++)      xrecord[i] = name[i];
  for (i=nam_len; i<23; i++)         xrecord[i] = ' ';
  for (i = 23; i < 23+tel_len; i++) xrecord[i] = tele[i-23];
  for (i=23+tel_len; i<26; i++)      xrecord[i] = ' ';
  for (i = 26; i < 26+car_len; i++) xrecord[i] = car_no[i-26];
  for (i=26+car_len; i<38; i++)      xrecord[i] = ' ';
  status[0] = '0'; status[1] = '0';
  ispwrt("SU", 0, 1, &xrecord[0], 0, 37, &status[0], 0, 1);
  if (strncmp(&status[0], "00", 2)) {
   print_error(0, "write.                          ");
   return(-1);
  }
 } while (nam_len);
 return(0);
}  /* newrec */
```

```
int liste(which)
{
 int i;
 short j;
 char c;
 struct {
         char name[23];
         char tele[3];
         char car_no[12];
        } frecord;
 if (strncmp(&status[0], "00", 2) == 0) {
  j = 0;
  fpdecfn("LIST", 0, 3, 0, &foc_stat);
  if (foc_stat) {
   print_error(1, "declare form name.               ");
   return(-1);
  }
  while (strncmp(&status[0], "00", 2) == 0) {
   isprnx("SU", 0, 1, &xrecord[0], 36, &status[0], 0, 1);
   if (strncmp(&status[0], "00", 2)) {
    status[0] = '0'; status[1] = '0';
    break;
   }
   if (which == 1) fpwfld("CHOICE", 0, 5, 1, "NAMES", 4, 5, &foc_stat);
   else
   if (which == 2) fpwfld("CHOICE", 0, 5, 1, "PHONE NUMBERS", 12, 13, &foc_stat);
   else
   if (which == 3) fpwfld("CHOICE", 0, 5, 1, "CAR REG. NO.", 11, 12, &foc_stat);
   if (foc_stat) {
    print_error(1, "write field(CHOICE).            ");
    return(-1);
   }
   for (i = 0; i < 23; i++) frecord.name[i] = xrecord[i];
   for (i = 23; i < 26; i++) frecord.tele[i - 23] = xrecord[i];
   for (i = 26; i < 38; i++) frecord.car_no[i - 26] = xrecord[i];
   j++;
   fldecrc("NAME PHONE CARNO'", 0, 16, j, j, &foc_stat);
   if (foc_stat) {
    print_error(1, "declare record.                 ");
    return(-1);
   }
   flwsub("NAME PHONE CARNO'", 0, 16, j, j, &frecord, sizeof(frecord)-1, &foc_stat);
   if (foc_stat) {
    print_error(1, "write subrecord.                ");
    return(-1);
   }
   if (j == 15) {
    j = 0;
    fpgmsge("Press <CR> to continue. ", 0, 23, &c, 0, 0, &foc_stat);
   }
  } /* while */
```

```
  if (j < 15) {
   for (i = 0; i < 23; i++) frecord.name[i] = ' ';
   for (i = 23; i < 26; i++) frecord.tele[i - 23] = ' ';
   for (i = 26; i < 38; i++) frecord.car_no[i - 26] = ' ';
   for (i=j+1; i<16; i++) {
    fldecrc("NAME PHONE CARNO'", 0, 16, i, i, &foc_stat);
    if (foc_stat) {
     print_error(1, "declare record.                  ");
     return(-1);
    }
    flwsub("NAME PHONE CARNO'", 0, 16, i, i, &frecord, sizeof(frecord)-1, &foc_stat);
    if (foc_stat) {
     print_error(1, "write (empty) subrecord.         ");
     return(-1);
    }
   }
  } /* if (j < 15) */
  fpgmsge("Press <CR> to return to the MAIN MENU. ", 0, 38, &c, 0, 0, &foc_stat);
 } /* if no ISAM-error */
 else {
  print_error(0, "start.                          ");
  return(-1);
 }
 return(0);
}  /* liste */

int namelist()
{
 short i;
 fct = 3;                                       /* find record with key greater than given value */
 memset(&xrecord[0], ' ', 38);                            /* set xrecord[] containing space only */
                                    /* If (and only if) the record starts with the key,         */
                                    /* you may use the isptrtl function for searching, i.e. the  */
                                    /* search key is passed as parameter to the function,        */
                                    /* which copies the key to the beginning of the record.      */
 isptrtl("SU", 0, 1, fct, "NA", 0, 1, &name_search[0], 23, &xrecord[0], 38, &status[0], 0, 1);
 if (liste(1)) return(-1);
 return(0);
}  /* namelist */

int telelist()
{
 short i;
 fct = 3;
                                 /* As the record does not start with the phone number as key      */
                                 /* you have to copy the search key to the right place             */
                                 /* within the record, and call the "normal" ISAM function isptrt. */
 memset(&xrecord[0], ' ', 38);
 for (i=23; i<26; i++) xrecord[i] = tele_search[23-i];
 isptrt("SU", 0, 1, fct, "TE", 0, 1, &xrecord[0], 38, UNUSED, &status[0], 0, 1);
 if (liste(2)) return(-1);
 return(0);
} /* telelist */
```

```
int carlist()
{
 short i;
 fct = 3;
 memset(&xrecord[0], ' ', 38);
 for (i=26; i<38; i++) xrecord[i] = car_search[26-i];
 isptrt("SU", 0, 1, fct, "CA", 0, 1, &xrecord[0], 38, UNUSED, &status[0], 0, 1);
 if (liste(3)) return(-1);
 return(0);
} /* carlist */

int del()                                                        /* delete record(s) */
{
 int c;
 short i;
 short nam_len;
 fpdecfn("DEL-REC", 0, 6, 0, &foc_stat);
 if (foc_stat) {
  print_error(1, "form name declaration(DEL-REC). ");
  return(-1);
 }
 fldecrc("NAME'", 0, 4, 1, 1, &foc_stat);
 if (foc_stat) {
  print_error(1, "record declaration.              ");
  return(-1);
 }

 do
 {
  fpefld("NAME", 0, 3, 1, &name[0], 22, &nam_len, 1, &foc_stat);
  if (foc_stat) {
   print_error(1, "edit one field(name to delete). ");
   return(-1);
  }
  if (nam_len == 0) break;
  for (i = 0; i < nam_len; i++)  xrecord[i] = name[i];
  for (i = nam_len; i < 38; i++) xrecord[i] = ' ';
  status[0] = '0'; status[1] = '0';
  ispdlk("SU", 0, 1, &xrecord[0], 38, UNUSED, &status[0], 0, 1);

  if (strncmp(&status[0], "00", 2) == 0) fpzmsge("Deleted!", 0, 7, &foc_stat);
  else {
   if ((strncmp(&status[0], "23", 2) == 0) && (nam_len))
    fpzmsge("Cannot find that name!", 0, 21, &foc_stat);
   else if (strncmp(&status[0], "23", 2)) {
    print_error(0, "file.                            ");
    return(-1);
   }
  }
 } while (nam_len);
 return(0);
}  /* del */

int mask()
{
  int i;
  char choice;
```

```
int mask()
{
  int i;
  char choice;
  char x;

  i = 0; choice = '\0';
  fpdecfn("MENU", 0, 3, 0, &foc_stat);
  if (foc_stat) {
   print_error(1, "form name declaration.          ");
   return(-1);
  }
  fldecrc("FUNCNO'", 0, 6, 1, 1, &foc_stat);
  if (foc_stat) {
   print_error(1, "declare record.                 ");
   return(-1);
  }
  attr = SLOW_BLINK;
  flsetat(&attr, "FUNCNO'", 0, 6, 1, 1, &foc_stat);
  if (foc_stat) {
   print_error(1, "set attributes.                 ");
   return(-1);
  }
  flsetmr("FUNCNO'", 0, 6, 1, 1, &foc_stat);
  if (foc_stat) {
   print_error(1, "set \"must read\".              ");
   return(-1);
  }
  fpefld("FUNCNO", 0, 5, 1, &choice, 0, 1, 1, &foc_stat);
  if (foc_stat) {
   print_error(1, "edit one field(CHOICE).         ");
   return(-1);
  }
  switch (choice)
  {
    case '1' : i=newrec();   break;
    case '2' : i=namelist(); break;
    case '3' : i=telelist(); break;
    case '4' : i=carlist();  break;
    case '5' : i=del();      break;
    case '9' : i=9;          break;
  } /* case */
  return(i);
} /* mask */
```

FOCUS INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
F1CLMR	1. &field_list 2. 0 3. strlen(field_list)-1 4. first_occ_number 5. last_occ_number 6. &status	string short short short	clear must read
F1CLSUB	1. &field_list 2. 0 3. strlen(field_list)-1 4. first_occ_number 5. last_occ_number 6. &data_record 7. sizeof(data_record) 8. &status	string short short any type short	clear subrecord
F1DECRC	1. &field_list 2. 0 3. strlen(field_list)-1 4. first_occ_number 5. last_occ_number 6. &status	string short short short	declare record
F1ESUB	1. &field_list 2. 0 3. strlen(field_list)-1 4. first_occ_number 5. last_occ_number 6. &data_record 7. sizeof(data_record) 8. edit_mode 9. &status	string short short any type short short	edit subrecord
F1FLDNA	1. from_field_number 2. to_field_number 3. &field_names 4. 0 5. sizeof(field_names)-1 6. &occ_numbers 7. 0 8. sizeof(occ_numbers)/2-1 9. &status	short short string short_array short	get field name

FOCUS INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
F1GSUB	1. &field_list 2. 0 3. strlen(field_list)-1 4. first_occ_number 5. last_occ_number 6. &data_record 7. sizeof(data_record) 8. &sub_data_record 9. sizeof(sub_data_record) 10. &status	string short short any type any type short	get data elements from a data record
F1PSUB	1. &field_list 2. 0 3. strlen(field_list)-1 4. first_occ_number 5. last_occ_number 6. &data_record 7. sizeof(data_record) 8. &sub_data_record 9. sizeof(sub_data_record) 10. &status	string short short any type any type short	put data elements into a data record
F1SETAT	1. attribute_code 2. &field_list 3. 0 4. strlen(field_list)-1 5. first_occ_number 6. last_occ_number 7. &status	attributes string short short short	set attributes (intensity, blink, inverse, etc)
F1SETMR	1. &field_list 2. 0 3. strlen(field_list)-1 4. first_occ_number 5. last_occ_number 6. &status	string short short short	set must read
F1WDOTS	1. &field_list 2. 0 3. strlen(field_list)-1 4. first_occ_number 5. last_occ_number 6. &status	string short short short	write dots in field

FOCUS INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
F1WSUB	1. &field_list 2. 0 3. strlen(field_list)-1 4. first_occ_number 5. last_occ_number 6. &data_record 7. **sizeof**(data_record) 8. &status	string **short** **short** any type **short**	write subrecord
FPBELL	1. &status	**short**	bell
FPCHRBR	1. nr_of_keys 2. &key_values 3. 0 4. **sizeof**(key_values)/2-1 5. &function_numbers 6. 0 7. **sizeof**(function_numbers)/2-1 8. &status	**short** short_array short_array **short**	define function keys
FPCLFDS	1. mode 2. &status	**short** **short**	clear fields
FPCLFO	1. &status	**short**	clear form
FPCLOSE	1. file_number 2. &status	**short** **short**	close file
FPCLREC	1. &data_record 2. **sizeof**(data_record) 3. &status	any type **short**	clear data record
FPCLSCR	1. from_line 2. from_column 3. to_line 4. to_column 5. &status	**short** **short** **short** **short** **short**	clear rectangular area on screen
FPDECFF	1. &form_file_name 2. 0 3. strlen(form_file_name)-1 2. &status	string **short**	declare form file
FPDECFN	1. &form_name 2. 0 3. strlen(form_name)-1 4. mode 5. &status	string **short** **short**	declare form name

FOCUS INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
FPDESB	1. &edit_status_buffer	status_struct	define edit status buffer
	2. **sizeof**(edit_status_buffer)		
	3. number_of_elements	**short**	
	4. &status	**short**	
FPEBUF	1. &status	**short**	empty buffer
FPEDSTA	1. &nr_fields_edited	**short**	get edit status
	2. &record_status	short_array	
	3. 0		
	4. **sizeof**(record_status)/2-1		
	5. &termination_code	**short**	
	6. &status	**short**	
FPEFLD	1. &field_name	string	edit one field
	2. 0		
	3. strlen(field_name)-1		
	4. occ_number	**short**	
	5. &data_element	any type	
	6. **sizeof**(data_element)		
	7. &data_length	**short**	
	8. edit_mode	**short**	
	9. &status	**short**	
FPEREC	1. &data_record	any type	edit record
	2. **sizeof**(data_record)		
	3. edit_mode	**short**	
	4. &status	**short**	
FPESFLD	1. &field_name	string	set edit start field
	2. 0		
	3. strlen(field_name)-1		
	4. occ_number	**short**	
	5. &status	**short**	
FPESFNC	1. fct_number	**short**	define edit start function
	2. line	**short**	
	3. column	**short**	
	4. &status	**short**	
FPFLDBR	1. nr_of_leavings	**short**	define leaving field functions
	2. VAR function_numbers	short array	
	3. 0		
	4. **sizeof**(function_numbers)/2-1		
	5. &status	**short**	

FOCUS INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
FPFLDIN	1. &field_name 2. 0 3. strlen(field_name)-1 4. occ_number 5. &field_info_array 6. **sizeof**(field_info_array) 7. field_info_length 8. &status	string **short** any type **short** **short**	get field information
FPFLDST	1. &field_name 2. 0 3. strlen(field_name)-1 4. occ_number 5. &field_status_array 6. **sizeof**(field_status_array) 7. field_status_length 8. &status	string **short** any type **short** **short**	get field status
FPFRMBR	1. nr_of_leavings 2. &function_numbers 3. 0 4. **sizeof**(function_numbers)/2-1 5. &status	**short** short_array **short**	define leaving form functions
FPFRMIN	1. &form_info_array 2. 0 3. **sizeof**(form_info_array)/2-1 4. info_array_length 5. &status	short_array **short** **short**	get form information
FPGLINE	1. from_line 2. to_line 3. &data_record 4. **sizeof**(data_record) 5. &buffer_area 6. **sizeof**(buffer_area) 7. &status	**short** **short** any type any type **short**	get form line
FPGMSGE	1. &leading_text 2. 0 3. strlen(leading_text)-1 4. &message 5. 0 6. **sizeof**(message)-1 7. &status	string string **short**	get message

FOCUS INTERFACE			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
FPINITE	1. &initiation_array 2. 0 3. **sizeof**(initiation_array)/2-1 4. &form_buffer 5. **sizeof**(form_buffer)-1 6. &status	short_array any type **short**	initiate and terminate FOCUS
FPNXFLD	1. &field_name 2. 0 3. strlen(field_name)-1 4. occ_number 5. position 6. &status	string **short** **short** **short**	define edit start field
FPOPEN	1. &file_name 2. 0 3. strlen(file_name)-1 4. &access_code 5. 0 6. strlen(access_code)-1 7. &file_number 8. &status	string string **short** **short**	open file
FPPCUR	1. line 2. column 3. &status	**short** **short** **short**	position cursor
FPPOSFO	1. line 2. column 3. &status	**short** **short** **short**	position form
FPPRDOC	1. &data_record 2. **sizeof**(data_record) 3. file_number 4. &status	any type **short** **short**	print document on file
FPRCHR	1. &input_char 2. &status	**char** **short**	read character
FPRDFO	1. mode 2. &data_record 3. **sizeof**(data_record) 4. &status	**short** any type **short**	redisplay form

F O C U S I N T E R F A C E			
NAME	PARAMETER	PARAMETER TYPE	DESCRIPTION
FPRTXT	1. line 2. column 3. &string 4. 0 5. **sizeof**(string)-1 6. &string_length 7. &status	short short string short short	read text
FPSAVFO	1. redisplay_flag 2. &data_record 3. **sizeof**(data_record) 4. &status	short any type short	save form
FPSCRIN	1. &screen_info_array 2. **sizeof**(screen_info_array) 3. screen_info_length 4. &status	any type short short	get screen information
FPWFLD	1. &field_name 2. 0 3. strlen(field_name)-1 4. occ_number 5. &data_elem 6. **sizeof**(data_elem) 7. data_length 8. &status	string short any type short short	write one field
FPWLTXT	1. mode 2. &status	short short	write leading texts
FPWREC	1. &data_record 2. **sizeof**(data_record) 3. &status	any type short	write record
FPWTXT	1. line 2. column 3. &string 4. 0 5. strlen(string)-1 6. &string_length 7. &status	short short string short short	write text
FPZMSGE	1. &message 2. 0 3. strlen(message)-1 4. &status	string short	send message

Chapter 16

Appendix A: ASCII character set

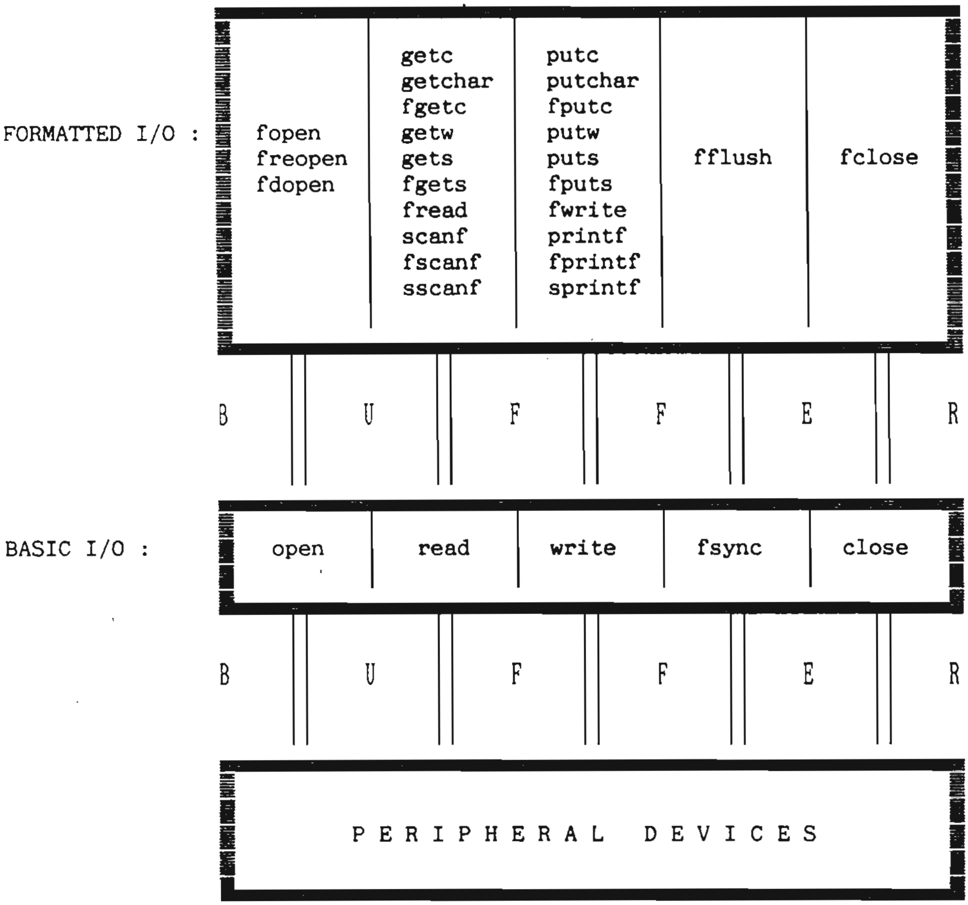
CHAR	DEC	OCT	HEXA	BINARY	CHAR	DEC	OCT	HEXA	BINARY	CHAR	DEC	OCT	HEXA	BINARY
NUL	0	0	00	0000 0000	+	43	53	2B	0010 1011	V	86	126	56	0101 0110
SOH	1	1	01	0000 0001	,	44	54	2C	0010 1100	W	87	127	57	0101 0111
STX	2	2	02	0000 0010	-	45	55	2D	0010 1101	X	88	130	58	0101 1000
ETX	3	3	03	0000 0011	.	46	56	2E	0010 1110	Y	89	131	59	0101 1001
EOT	4	4	04	0000 0100	/	47	57	2F	0010 1111	Z	90	132	5A	0101 1010
ENQ	5	5	05	0000 0101	0	48	60	30	0011 0000	[	91	133	5B	0101 1011
ACK	6	6	06	0000 0110	1	49	61	31	0011 0001	\	92	134	5C	0101 1100
BEL	7	7	07	0000 0111	2	50	62	32	0011 0010	]	93	135	5D	0101 1101
BS	8	10	08	0000 1000	3	51	63	33	0011 0011	^	94	135	5E	0101 1110
HT	9	11	09	0000 1001	4	52	62	34	0011 0100	_	95	137	5F	0101 1111
LF	10	12	0A	0000 1010	5	53	65	35	0011 0101	`	96	140	60	0110 0000
VT	11	13	0B	0000 1011	6	54	66	36	0011 0110	a	97	141	61	0110 0001
FF	12	14	0C	0000 1100	7	55	67	37	0011 0111	b	98	142	62	0110 0010
CR	13	15	0D	0000 1101	8	56	70	38	0011 1000	c	99	143	63	0110 0011
SO	14	16	0E	0000 1110	9	57	71	39	0011 1001	d	100	144	64	0110 0100
SI	15	17	0F	0000 1111	:	58	72	3A	0011 1010	e	101	145	65	0110 0101
DLE	16	20	10	0001 0000	;	59	73	3B	0011 1011	f	102	146	66	0110 0110
DC1	17	21	11	0001 0001	<	60	74	3C	0011 1100	g	103	147	67	0110 0111
DC2	18	22	12	0001 0010	=	61	75	3D	0011 1101	h	104	150	68	0110 1000
DC3	19	23	13	0001 0011	>	62	76	3E	0011 1110	i	105	151	69	0110 1001
DC4	20	24	14	0001 0100	?	63	77	3F	0011 1111	j	106	152	6A	0110 1010
NAK	21	25	15	0001 0101	@	64	100	40	0100 0000	k	107	153	6B	0110 1011
SYN	22	26	16	0001 0110	A	65	101	41	0100 0001	l	108	154	6C	0110 1100
ETB	23	27	17	0001 0111	B	66	102	42	0100 0010	m	109	155	6D	0110 1101
CAN	24	30	18	0001 1000	C	67	103	43	0100 0011	n	110	156	6E	0110 1110
EM	25	31	19	0001 1001	D	68	104	44	0100 0100	o	111	157	6F	0110 1111
SUB	26	32	1A	0001 1010	E	69	105	45	0100 0101	p	112	160	70	0111 0000
ESC	27	33	1B	0001 1011	F	70	106	46	0100 0110	q	113	161	71	0111 0001
FS	28	34	1C	0001 1100	G	71	107	47	0100 0111	r	114	162	72	0111 0010
GS	29	35	1D	0001 1101	H	72	110	48	0100 1000	s	115	163	73	0111 0011
RS	30	36	1E	0001 1110	I	73	111	49	0100 1001	t	116	164	74	0111 0100
US	31	37	1F	0001 1111	J	74	112	4A	0100 1010	u	117	165	75	0111 0101
SPC	32	40	20	0010 0000	K	75	113	4B	0100 1011	v	118	166	76	0111 0110
!	33	41	21	0010 0001	L	76	114	4C	0100 1100	w	119	167	77	0111 0111
"	34	42	22	0010 0010	M	77	115	4D	0100 1101	x	120	170	78	0111 1000
#	35	43	23	0010 0011	N	78	116	4E	0100 1110	y	121	171	79	0111 1001
$	36	44	24	0010 0100	O	79	117	4F	0100 1111	z	122	172	7A	0111 1010
%	37	45	25	0010 0101	P	80	120	50	0101 0000	{	123	173	7B	0111 1011
&	38	46	26	0010 0110	Q	81	121	51	0101 0001	|	124	174	7C	0111 1100
'	39	47	27	0010 0111	R	82	122	52	0101 0010	}	125	175	7D	0111 1101
(	40	50	28	0010 1000	S	83	123	53	0101 0011	~	126	176	7E	0111 1110
)	41	51	29	0010 1001	T	84	124	54	0101 0100	DEL	127	177	7F	0111 1111
*	42	52	2A	0010 1010	U	85	125	55	0101 0101					

Norsk Data ND-860251.2 EN

Chapter 17

Appendix B: The I/O System

THE I/O SYSTEM



Chapter 18

Appendix C: List of Functions

Norsk Data ND-860251.2 EN

INDEX LIST

Index term	Reference

Index term | Reference

Index term	Reference

Index term	Reference

Index term	Reference