Par.C System

User's Manual and Library Reference

Version 1.22

4th edition, December 18, 1989

Information in this document is subject to change without notice and does not represent a commitment on the part of Parsec Developments. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. The purchaser may make one copy of the software for backup purposes. No part of this publication or its accompanying software may be sold, resold, given away or reproduced or transmitted in any form or by any means, mechanical or electronic, including recording and photocopying, for any purpose other than the purchaser's personal use without the prior written permission of Parsec Developments.

Copyright © Parsec Developments, 1987-1990. All rights reserved.

You may sell or distribute software written with this version of the Par.C System freely when you mention that it was made with the Par.C System of Parsec Developments. You may also let your software be accompanied with one copy of our server (RUN) and our resident libraries (*.RLS).

Parsec Developments, the Parsec logo, Par.C and the Par.C System are registered trademarks of Parsec Developments. Other brand and product names are trademarks or registered trademarks of their respective holders.

If you have any comments about the software or the manual, please contact Parsec Developments. If you come across an error either in the software or the manual, please complete the form at the back of this manual and return it to Parsec Developments.

The Par.C System and its manual were written by Jan de Baat, Frans Evenblij, Jang Graat, Erik Groenhuis, Jeanette Hooimeyer, Martien de Jong, Frits Kuijlman, Lex Lissauer, Leo Noordergraaf, Hans Onvlee, Maurice van Peursem and Ko Stoel.

Table of Contents

Introduction

Introduction

The Par.C System is more than yet another C compiler for Inmos transputers: the system has been carefully designed to match the transputer hardware in flexibility, and the result provides a powerful programming environment on the basis of a parallelisation of the language C.

Parallel C is a superset of C, containing the main facilities for parallel programming in the **par** and **select** statements and the **channel** datatype. The extensions resemble the main features in Inmos's OCCAM, but have been implemented in C-style, without interfering with the rules of the existing language. This has resulted in an easy-to-use, flexible and powerful language for parallel programming.

The Par.C System runtime libraries support the flexibility in the language to a maximum extent. Many ANSI C functions have been implemented to be re-entrant and various special functions have been added to access specific features of transputer clusters.

The unique Par.C boot System can handle transputer networks of any size and topology, and passes information on the available hardware to the program that is loaded. On the basis of this feature, software can easily be made to adjust itself to the available hardware during runtime. In this case, it is no longer necessary to recompile or relink a program after the configuration of the transputer cluster has been changed. Plug in an extra board and your program runs faster.

The Par.C System Manual not only provides a complete description of the different features of the software. Parts of the manual were added to introduce the main concepts concerning parallel programming, which have been used in the design of the Par.C System. The Tutorial chapter is an introduction to the Par.C System and is based on example programs included on disk. This chapter is followed by a Special Topics chapter for the advanced Par.C System user.

Parsec Developments is constantly working on improvements of the Par.C System and extended facilities. If you want to receive information on new features and updates of the system, please do not forget to return the registration form. If you have special demands or suggestions, we are always interested in hearing them. Contact your dealer or write:

Parsec Developments
P.O. Box 782, 2300 AT Leiden
The Netherlands
Phone: +31 71 142142
Fax: +31 71 134449

Intended audience

The Par.C System is aimed at users of the Inmos transputer family of microprocessors, either as a single chip or interconnected into networks of any size and configuration. No specific operating system is needed to use the Par.C System.

Since the system is based on a parallelisation of the language C, some knowledge of this programming language is assumed. Novice C users are advised to use one of the available C textbooks to learn the standard language, before moving on to the parallel extensions offered in the Par.C System. A good introduction is given in "The C programming language" by the founders of the C language, Kernighan and Ritchie [K&R].

Knowledge of the host system, on which the Par.C System runs and from which Par.C-produced programs are downloaded to the transputer network, is also assumed. Details on installation of the Par.C System on particular host machines, and on adaptation of the Par.C loader/server to different interfacing hardware, is given in the Installation chapter.

Hardware assumptions

The Par.C System has been produced to run on a variety of "host" hardware, including transputers. This section indicates the necessary environment in general, for Par.C produced programs and for the Par.C System itself. Details on the necessary configurations of the host machine for this particular version of the Par.C System are given in the chapter on Installation.

Target hardware requirements

The Par.C System generates code for the Inmos T4xx and T8xx transputers, interconnected into clusters of any size, mixture and topology. The minimum amount of external memory needed per transputer is 0 (zero) KBytes (except when using the Par.C native System). Of course, the amount of memory needed in practice depends on the Par.C produced programs that should run on the transputer cluster.

Transputer boards from any hardware supplier are supported, insofar as the transputers are interconnected through their links, and the network can be booted through one single linkconnection with the host machine. The Par.C loader/server can be adjusted to different hardware interfaces by changing the parameter settings in `hardware.cnf`.

The Par.C bootsystem expects all transputers in the target system to be in a reset state. 'Active' transputers will not be found, but errors may occur in the software running on them, as a result of the investigation efforts of the Par.C bootsystem.

Note .... An adaptation to the Parsytec "nearest neighbour" reset scheme has been made, in which the bootsystem actively resets the entire cluster while investigating it.

Requirements for the Par.C native System

The Par.C native System runs on the transputer system. The only part of the system running on the host machine in this version is the Par.C loader/server, which downloads the compiler, assembler and linker to the transputers, just like it does with user-written programs.

With the Par.C native System, the minimum amount of memory needed on the first transputer in the system is 256 KBytes. However, when producing large programs, this size will be insufficient.

Requirements for the Par.C cross System

The Par.C cross System is available on various hosts e.i. IBM PC, Sun, Apollo etc. The size of the available memory determines the size of the programs you will be able to produce. For details on the necessary version of host operating system and possible extras needed for your machine, see the Installation chapter of this manual.

Available versions of the Par.C System

The Par.C System is available in a number of versions. These are introduced with the objective of offering users with restricted budgets the advantages of working with the Par.C System. Upgrades to the full system are available for all of the cheaper versions. The features of the different versions are described below.

1. Single T4 System

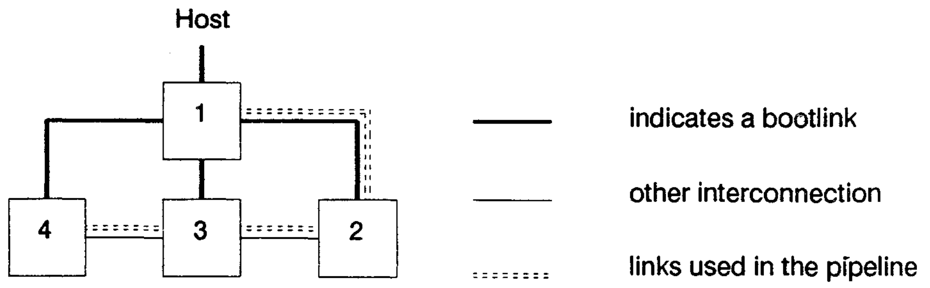
This is the cheapest version of the Par.C System, and is restricted in that the code is only generated for one single T4 transputer. If the code does not contain floating point calculations, it will also run on a single T8 transputer (the T8 may crash on instructions which are used on the T4 for floating point emulation).

2. Single T8 System

This version generates code for one single T8 transputer, using the special T8math library for floating point calculations. If the code does not contain floating point calculations, it will also run on a single T4 transputer.

3. Small System

This version generates code for single transputers, but includes the possibility of through-booting to either of the links of the first node. This results in a maximum target system size of four transputers, configured in a tree topology. Note that the links which are not in the boottree but are connected, still make it possible to have a pipeline-application running on this four-node system.



A pipeline in a Small System

4. Static System

This version allows the user to generate single-transputer tasks, which can be placed on processors throughout the entire network. This model looks most like the configuration models used in various other development systems. The difference with these systems is that there is no configurer combining separately compiled loadmodules with bootcode to obtain one single binary image to download. Instead, a script is given to the loader, containing a list of loadmodules and destinations. Since only single-transputer tasks can be produced, at least one task has to be specified for each processor. In contrast to the small system, there are no restrictions to the size of the network to run the program on.

5. Dynamic System

This version contains all features of the Par.C System. The difference with all the above systems is, that programs for multiple processors can be written, and these programs can be made to adjust themselves to the hardware each time they are executed.

Note .... Some of our distributors may not offer all versions listed.

About this Par.C System manual

This manual is part of the Par.C System. It consists of several chapters, each numbered independently of other chapters. Page numbers are preceded by the chapter number. A range is indicated with dots separating the (included) boundaries, as shown below:

4-5: indicates page 5 of chapter 4

1-2..6: indicates pages 2 through 6 of chapter1

Subdivision into chapters

The manual consists of several chapters containing information about specific subjects of the Par.C System. At the end of the manual an index is given with references for the keywords occuring in the complete manual. A short description of the chapters is given below:

Quick Reference . The Quick Reference chapter provides fast access to the definitions of the most important features of the Par.C System.

This part contains :

- Syntax of language extensions
- Library functions with prototypes
- Implementation details concerning this Par.C System
- List of definitions for most of the important terms used

Installation ...... This chapter contains a description of the contents of the Par.C System package. It further explains the installation procedure for your host machine, followed by a confidence test checking the system installed.

The Par.C System This chapter gives an overview of the Par.C System. The use of the constituant parts is explained, followed by a description of the Par.C Runtime System.

Parallel C .... The extensions to the standard C language are described in this chapter. The **par** statement, the **channel** datatype and the **select** statement are explained here.

Tutorial .... The tutorial is meant to give some examples and indications about writing parallel programs using the Par.C System. Several types of programs will be discussed, ranging from single transputer programs to multi-transputer, network independent programs. This chapter also contains some demonstration programs written in the Par.C language.

Special Topics ... The special topics include some specific uses of the Par.C System, different ways to use the Par.C Runtime System and memory usage of the Par.C System. It also contains some tips and tricks about optimisation techniques and the configuration of transputer networks.

Utilities .... The chapter about the utilities includes explanations of a network displayer (`sysnet.run`). A stand-alone program builder (RUN2EXE) is also discussed.

Libraries .... The Par.C System libraries consist of standard C library functions conform the ANSI C standard, completed with extra library functions for the Par.C Runtime System. First an overview of the library functions is given, followed by a description of the header files declaring them. After that, a complete alphabetical list of runtime functions is given with a full explanation of their use and functionality.

Appendices ...... In the first appendix an overview is given of the differences with the previous version of the Par.C System. A list of known bugs and remedies is added. The next appendix contains a complete list of all error and warning messages that can be generated by the Par.C System. The messages are grouped together according to the part of the system they are generated by: compiler, assembler, linker or loader/server. Furthermore an ASCII table and the literature references are included.

Typographic and notational conventions

In the description of the syntax and language definition (chapter on Parallel C), the following typographic and notational conventions will be used:

In the examples, identifiers are chosen in relation to their use for explanatory reasons. These names can be replaced by any valid identifier. In listings of examples, code which bears no significance for what the example tries to illustrate is replaced by one or more lines with a sequence of dots.

When showing examples of parallel C source code comments are enclosed in /* and */ as usual. Indentation is used to indicate the scope of each compound statement. In comments on examples, the term "level" is used in accordance with the scope of the code that is commented. The level number agrees with the level of indentation in the source shown. Level 0 is defined to be the function level.

In some of the explanations, transputer assembly language mnemonics are used. In this assembly source code, comments start with a semicolon and end with the next newline. Indentation is used to make a distinction between labels, instructions and operands. Labels are placed at the extreme left side of a new line. The transputer assembly language mnemonics are placed at the first level of indentation. The operands used in the instructions are placed on the second level.

Further reading

This manual is not intended for learning the standard C language or giving insight to parallel programming. For these matters the reader is refered to the list below.

The C programming language

- Kernighan and Ritchie [K&R, 1978]
- Harbison and Steele [HARBISON, 1984]

Parallel programming

- C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985
- R. Perrott, *Parallel programming*, Addison-Wesley, 1987
- M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, 1987

Quick reference

Syntax of language extensions

Any text below which is set in **boldface** is called a terminal and stands for text as you would type it in. The rest of the text which is not set in boldface is called a non-terminal and stands for a rule you should look up below or in a standard C grammar. If text is enclosed in square brackets (*'['* and *']'*) it means that it is optional. Text followed by a '+' can occur 1 or more times. A non-terminal in the left column can be followed by more than one rule. Each of these rules stands for one alternative by which the non-terminal can be replaced.

non-terminal	**rule**
par_statement:	**par** *[replicator]* { *statement* + }
type_specifier:	**channel**

select_statement:	**select** { *alt_section* + }
	select within *expression* { *timed_alt_section* + }

alt_section:	*guard_alternative* : *statement_list* *default_alternative* : *statement_list*
timed_alt_section:	*alt_section* *timeout_alternative* : *statement_list*
guard_alternative:	**alt** *[replicator] [***cond** *expression]* **guard** *expression*
default_alternative:	**alt** *[replicator]* **cond** *expression*
timeout_alternative:	**alt timeout** *[***cond** *expression]*
replicator:	(*expression* ; *expression* ; *expression*)

Runtime functions

All library runtime functions with their prototypes:

Note: prototypes are an ANSI C feature. They are not supported by the Par.C System, but are included here to clearify the use of the functions.

type	function	function/macro	defined in
	a		
void	abort (void)	function	stdlib.h
void	_abort (void)	function	stdlib.h
int	abs (int x)	function	stdlib.h
double	acos (double x)	function	math.h
int	AFTER (time_t t1, time_t t2)	macro	time.h
unsigned	alarm (int n)	function	time.h
clock_t	_alarm (clock_t ticks)	function	time.h
char *	asctime (struct tm *ptr)	function	time.h
char *	asctime_r (struct tm *ptr, char *s)	function	time.h
double	asin (double x)	function	math.h
void	assert (int expr)	macro	assert.h
double	atan (double x)	function	math.h
double	atan2 (double y, double x)	function	math.h
int	atexit (void(*fptr)(void))	function	stdlib.h
uns long	_ato (char *str, int cntrl,[char **output])	function	stdcnv.h
unsigned	atob (char *s)	macro	stdcnv.h
uns long	atobl (char *s)	macro	stdcnv.h
double	atod (char *s)	macro	stdcnv.h
double	atof (char *s)	function	stdlib.h
double	atof (char *s)	macro	stdcnv.h
int	atoi (char *s)	function	stdlib.h
int	atoi (char *s)	macro	stdcnv.h
long	atol (char *s)	function	stdlib.h
long	atol (char *s)	macro	stdcnv.h
unsigned	atoo (char *s)	macro	stdcnv.h
uns long	atool (char *s)	macro	stdcnv.h
unsigned	atou (char *s)	macro	stdcnv.h
uns long	atoul (char *s)	macro	stdcnv.h
unsigned	atox (char *s)	macro	stdcnv.h
uns long	atoxl (char *s)	macro	stdcnv.h

type	function	function/macro	defined in
	b		
int	BEFORE (time_t t1, time_t t2)	macro	time.h
char *	bsearch (char *key, char *base, size_t n, size_t size, int (*comp)(char *p, char *q))	function	stdlib.h
int	btoa (int i, char *s)	macro	stdcnv.h
int	bltoa (long l, char *s)	macro	stdcnv.h
	c		
char *	calloc (size_t n, size_t size)	function	stdlib.h
double	ceil (double x)	function	math.h
void	clearerr (FILE *stream)	function	stdio.h
clock_t	clock (void)	function	time.h
clock_t	_ClrHaltErr (void)	function	transp.h
double	cos (double x)	function	math.h
double	cosh (double x)	function	math.h
char *	ctime (time_t *timeptr)	function	time.h
	d		
int	DateToInt (int day, int month, int year)	function	time.h
void	delay (int n)	function	time.h
void	Deschedule (void)	function	transp.h
double	difftime (time_t time1, time_t time0)	function	time.h
DIR *	Dir (DIR *search)	function	stdio.h
DIR *	DirInit (char *filedesc)	function	stdio.h
int	dtoa (double d, char *s)	macro	stdcnv.h
	e		
void	(*ERROR)(int errno, char *mess, ...)	function	errno.h
void	exit (int status)	function	stdlib.h
void	_exit (int status)	function	stdlib.h
double	exp (double x)	function	math.h
	f		
double	fabs (double x)	function	math.h
int	fclose (FILE *stream)	function	stdio.h
int	feof (FILE *stream)	macro	stdio.h
int	ferror (FILE *stream)	function	stdio.h
int	fflush (FILE *stream)	function	stdio.h
int	fgetc (FILE *stream)	function	stdio.h
char *	fgets (char *s, int n, FILE *stream)	function	stdio.h
int	filerr (FILE *stream)	macro	stdio.h

type	function	function/macro	defined in
double	floor (double x)	function	math.h
double	fmod (double x, double y)	function	math.h
FILE *	fopen (char *filename, char *mode)	function	stdio.h
int	fprintf (FILE *stream, char *format, ...)	function	stdio.h
int	fputc (int c, FILE *stream)	function	stdio.h
int	fputs (char *s, FILE *stream)	function	stdio.h
size_t	fread (char *dest, size_t size, size_t count, FILE *stream)	function	stdio.h
int	free (char *ptr)	function	stdlib.h
double	frexp (double x, int *exp)	function	math.h
int	fscanf (FILE *stream, char *format, ...)	function	stdio.h
int	fseek (FILE *stream, long int offset, int ref)	function	stdio.h
long int	ftell (FILE *stream)	function	stdio.h
int	ftoa (char *s, double d)	function	stdcnv.h
size_t	fwrite (char *src, size_t size, size_t count, FILE *stream)	function	stdio.h

g

type	function	function/macro	defined in
int	getc (FILE *stream)	macro	stdio.h
int	getchar (void)	macro	stdio.h
char *	getenv (char *name)	function	stdlib.h
int	getenv_r (char *value, char *name)	function	stdlib.h
int	_GetFunStack (void)	function	system.h
int	_GetParStack (void)	function	system.h
int	GetNodeInfo (int Tn, NODEDESCRIPTOR * NodeDesc)	function	system.h
int	GetSysInfo (SYSTEM *system)	function	system.h
char *	gets (char *s)	function	stdio.h
struct tm *	gmtime (time_t *timer)	function	time.h
struct tm *	gmtime_r (time_t *timer, struct tm *ptr)	function	time.h

i

type	function	function/macro	defined in
void	_in (channel *from, char *mess, int size)	function	transp.h
void	InMess (char *mess, int size, channel *from)	function	system.h
void	InPort (int prtaddr, int size, char *buff)	function	stdio.h
int	IntToDate (int lindat, int *pday, int *pmonth, int *pyear)	function	time.h
int	isalnum (char c)	function	ctype.h
int	isalpha (char c)	function	ctype.h
int	isascii (char c)	function	ctype.h
int	iscntrl (char c)	function	ctype.h
int	isdigit (char c)	function	ctype.h
int	isgraph (char c)	function	ctype.h
int	islower (char c)	function	ctype.h

type	function	function/macro	defined in
int	isodigit (char c)	function	ctype.h
int	isprint (char c)	function	ctype.h
int	ispunct (char c)	function	ctype.h
int	isspace (char c)	function	ctype.h
int	isupper (char c)	function	ctype.h
int	iswhite (char c)	function	ctype.h
int	isxdigit (char c)	function	ctype.h
int	itoa (int i, char *s)	macro	stdcnv.h

l

long int	labs (long int x)	function	stdlib.h
double	ldexp (double x, int exp)	function	math.h
int	_LDHB (void)	function	transp.h
int	_LDHF (void)	function	transp.h
int	_LDLB (void)	function	transp.h
int	_LDLF (void)	function	transp.h
channel *	LINKIN (int LinkNo)	macro	transp.h
channel *	LINKOUT (int LinkNo)	macro	transp.h
struct tm *	localtime (time_t *timer)	function	time.h
struct tm *	localtime_r (time_t *timer, struct tm *ptr)	function	time.h
double	log (double x)	function	math.h
double	log10 (double x)	function	math.h
int	ltoa (long l, char *s)	macro	stdcnv.h

m

char *	malloc (size_t size)	function	stdlib.h
double	matherr (int code, _MATHARGS arglist)	function	math.h
double	_matherr (int code, _MATHARGS arglist)	function	math.h
MEMINFO*	MemAvail (MEMINFO *MemInfo)	function	stdlib.h
char *	memchr (*s, char c, size_t n)	function	string.h
int	memcmp (*obj1, char *obj2, size_t n)	function	string.h
char *	memcpy (char *dest, char *src, size_t n)	function	string.h
char *	memfill (char *obj, char *ptr, int size, int cnt)	function	string.h
char *	memmove (char *dest, char *src, size_t n)	function	string.h
char *	memset (char *obj, int c, size_t n)	function	string.h
time_t	mktime (struct tm *ptr)	function	time.h
double	modf (double x, double *dptr)	function	math.h

n

int	NaN (double d)	function	math.h

type	function	function/macro	defined in
	o		
size_t	offsetof (struct s_name, member m_name)	macro	stddef.h
int	otoa (int i, char *s)	macro	stdcnv.h
int	oltoa (long l, char *s)	macro	stdcnv.h
int	onexit (void(*fptr)(void))	macro	stdlib.h
void	_out (channel *to, char *mess, int size)	function	transp.h
void	OutMess (char *mess, int size, channel *to)	function	system.h
void	OutPort (int prtaddr, int size, char *buff)	function	stdio.h
	p		
void	P (semaphore s)	function	stddef.h
void	PeekHost (int memaddr, int size, char *buff)	function	stdio.h
void	perror (char *s)	function	stdio.h
void	PokeHost (int memaddr, int size, char *buff)	function	stdio.h
double	pow (double x, double y)	function	math.h
double	pow2 (double x)	function	math.h
double	pow10 (double x)	function	math.h
int	printf (char *format, ...)	function	stdio.h
int	Priority (void)	function	system.h
int	putc (int c, FILE *stream)	macro	stdio.h
int	putchar (int c)	macro	stdio.h
int	puts (char *s)	function	stdio.h
	q		
void	qsort (char *base, size_t count, size_t size, int (*compare)(char *p1, char *p2))	function	stdlib.h
	r		
int	raise (int sig)	function	signal.h
uns int	rand (void)	function	stdlib.h
char *	realloc (char *oldptr, size_t newsize)	function	stdlib.h
int	RecvLink (int LinkNo, char *MessPtr, int MessSize)	function	stdlib.h
int	RecvLinkOrFail (int LinkNo, char *MessPtr, int MessSize, clock_t TimeOut)	function	stdlib.h
void	Release (int status)	function	stdlib.h
int	remove (char *filename)	function	stdio.h
int	rename (char *oldname, char *newname)	function	stdio.h
ProcessDescriptor	ResetChannel (channel *ptr)	function	transp.h
void	_ResetSystemTimers (clock_t Start)	function	transp.h
int	_RestartProcess (ProcessDescriptor P)	function	transp.h

type	function	function/macro	defined in
void	rewind (FILE *stream)	macro	stdio.h
int	Run (int LinkNo, char *Progfilename, char *Arguments)	function	stdio.h
void	RunProcess (void (*Process)(...), int Priority, int Npar, ...)	function	stdlib.h
	S		
int	scanf (char *format, ...)	function	stdio.h
int	SendLink (int LinkNo, char *MessPt, int MessSizer)	function	stdlib.h
int	SendLinkOrFail (int LinkNo, char *MessPtr, int MessSize, clock_t TimeOut)	function	stdlib.h
void	setbuf (FILE *stream, char *buf)	function	stdio.h
int	_SetErr (void)	function	transp.h
int	_SetFunStack (int DefaultFuncStackSize)	function	system.h
void	_SetHaltErr (void)	function	transp.h
int	_SetParStack (int DefaultParStackSize)	function	system.h
int	SetPriority (int p)	function	system.h
int	setvbuf (FILE *stream, char *buf, int mode, size_t size)	function	stdio.h
void	SIG_DFL (int sig)	function	signal.h
void	SIG_IGN (int sig)	function	signal.h
void	SIG_ERR (int sig)	function	signal.h
void	(* signal(int sig, void (*fptr)(int sig))) (int sig)	function	signal.h
double	sin (double x)	function	math.h
double	sinh (double x)	function	math.h
unsigned	sleep (int n)	function	time.h
clock_t	_sleep (clock_t ticks)	function	time.h
char *	smalloc (size_t size)	function	stdlib.h
int	sprintf (char *s, char *format, ...)	function	stdio.h
double	sqrt (double x)	function	math.h
void	srand (unsigned x)	function	stdlib.h
int	sscanf (char *s, char *format, ...)	function	stdio.h
void	_STHB (ProcessDescriptor P)	function	transp.h
void	_STHF (ProcessDescriptor P)	function	transp.h
void	_STLB (ProcessDescriptor P)	function	transp.h
void	_STLF (ProcessDescriptor P)	function	transp.h
void	_StopProcess (void)	function	transp.h
char *	strcat (char *s1, char *s2)	function	string.h
char *	strchr (char *s, char c)	function	string.h
int	strcmp(char *s1, char *s2)	function	string.h
char *	strcpy(char *s1, char *s2)	function	string.h
int	strcspn(char *s, char *set)	function	string.h

type	function	function/macro	defined in
char *	strerror(int errnum)	function	string.h
int	strlen(char *s)	function	string.h
char *	strncat(char *s1, char *s2, int n)	function	string.h
int	strncmp(char *s1, char *s2, int n)	function	string.h
char *	strncpy (char *s1, char *s2, int n)	function	string.h
char *	strpbrk (char *s, char *set)	function	string.h
char *	strpcbrk (char *s, char *set)	function	string.h
char *	strrchr (char *s, char c)	function	string.h
char *	strrpbrk (char *s, char *set)	function	string.h
int	strrpos (char *s, char c)	function	string.h
int	strspn (char *s, char *set)	function	string.h
char *	strstr (char *src, char *sub)	function	string.h
double	strtod (char *s1, char **s2)	function	stdcnv.h
		function	stdlib.h
long int	strtol (char *s1, char **s2, int base)	function	stdlib.h
		macro	stdcnv.h
char *	strtok (char *s, char *set)	function	string.h
char *	strtok_r (char *s, char **state, char *set)	function	string.h
long int	strtoul (char *s1, char **s2, int base)	function	stdlib.h
		macro	stdcnv.h
int	system (char *command)	function	stdlib.h

t

double	tan (double x)	function	math.h
double	tanh (double x)	function	math.h
int	_TestErr (void)	function	transp.h
int	_TestClrErr (void)	function	transp.h
clock_t	_TicksPerSecond (void)	function	time.h
time_t	time (time_t *ptr)	function	time.h
int	_toa (char *string, int control, [unsigned] [long] int input)	function	stdcnv.h
int	toascii (int i)	macro	ctype.h
int	toint (char c)	function	ctype.h
int	tolower (char c)	function	ctype.h
int	_tolower (char c)	macro	ctype.h
int	toupper (char c)	function	ctype.h
int	_toupper (char c)	macro	ctype.h

u

int	ungetc (int c, FILE *stream)	function	stdio.h
int	utoa (int i, char *s)	macro	stdcnv.h
int	ultoa (long l, char *s)	macro	stdcnv.h

type	function	function/macro	defined in
	V		
void	V (semaphore S)	function	stddef.h
type	va_arg (va_list ap, type)	macro	stdarg.h
int	_va_arg (va_list ap, size_t size)	function	stdarg.h
void	va_end (va_list ap)	macro	stdarg.h
void	va_start (va_list ap, parm)	macro	stdarg.h
int	vfprintf (FILE *stream, char *format, va_list args)	function	stdio.h
int	vprintf (char *format, va_list args)	function	stdio.h
int	vsprintf (char *s, char *format, va_list args)	function	stdio.h
	W		
void	wait (clock_t ticks)	function	time.h
	X		
int	xtoa (int i, char *s)	macro	stdcnv.h
int	Xtoa (int i, char *s)	macro	stdcnv.h
int	xltoa (long l, char *s)	macro	stdcnv.h
int	Xltoa (long l, char *s)	macro	stdcnv.h
	Y		
int	yday (int day, int month, int year)	function	time.h

Macros

Macros defined in various header files

Macro	Defined in	Macro	Defined in
ABORT_SIG	signal.h	SEEK_END	stdio.h
ALARM_SIG	signal.h	LBUFSIZ	stdio.h
EVENT_SIG	signal.h	FNULL	stdio.h
LinksPerTransputer	stddef.h	RAND_MAX	stdlib.h
MostNeg	stddef.h	CLK_TCK	time.h
MostPos	stddef.h	CLOCKS_PER_SEC	time.h
BytesInWord	stddef.h	LOW_PRIORITY	transp.h
_IOFBF	stdio.h	HIGH_PRIORITY	transp.h
_IOLBF	stdio.h	EVENT	transp.h
_IONBF	stdio.h	NotProcessP	transp.h
BUFSIZ	stdio.h	NULL	stddef.h
EOF	stdio.h	TRUE	stddef.h
MAX_OPEN	stdio.h	FALSE	stddef.h
SEEK_SET	stdio.h	MinInt	stddef.h
SEEK_CUR	stdio.h		

ANSI predefined macros:

`__LINE__`	: replaced by the current linenumber in the sourcefile
`__FILE__`	: replaced by the current sourcefile name
`__DATE__`	: replaced by the current date
`__TIME__`	: replaced by the current time
`__STDC__`	: replaced by FALSE in this Par.C compiler

Par.C predefined macros

`__FUNC__`	: replaced by the current function name
`__RAND__`	: replaced by a random generated number
`__PARC__`	: defined if Par.C is used, replaced by 1

Compiler-dependent directives

#pragma fpu	: generate T8 specific fpu code
#pragma nofpu	: generate T4/T8 transparant floating point code (default)
#pragma noint	: generate non-interuptable code
#pragma int	: generate interuptable code (default)

Implementation of basic types

Sizes and ranges of the basic Par.C types

		macro	defined in
char			
number of bits:	8		
largest negative value:	-128	CHAR_MIN	limits.h
largest positive value:	127	CHAR_MAX	limits.h
short			
number of bits:	16		
largest negative value:	-32768	SHRT_MIN	limits.h
largest positive value:	32767	SHRT_MAX	limits.h
int			
number of bits:	32		
largest negative value:	-2147483648	INT_MIN	limits.h
largest positive value:	2147483647	INT_MAX	limits.h
long			
number of bits:	64		
largest negative value:	-9223372036854775808	LONG_MIN	limits.h
largest positive value:	9223372036854775807	LONG_MAX	limits.h
unsigned char			
number of bits:	8		
smallest value:	0		
largest value:	255	UCHAR_MAX	limits.h
unsigned short			
number of bits:	16		
smallest value:	0		
largest value:	65535	USHRT_MAX	limits.h
unsigned int			
number of bits:	32		
smallest value:	0		
largest value:	4294967295	UINT_MAX	limits.h

unsigned long

		macro	defined in
number of bits:	64		
smallest value:	0		
largest value:	18446744073709551615	ULONG_MAX	limits.h

float

		macro	defined in
number of bits:	32		
largest negative value:	$-3.402823466 \times 10^{38}$		
smallest negative value:	$-1.175494351 \times 10^{-38}$		
zero:	0.0		
smallest positive value:	$1.175494351 \times 10^{-38}$	FLT_MIN	float.h
largest positive value:	$3.402823466 \times 10^{38}$	FLT_MAX	float.h

double

		macro	defined in
number of bits:	64		
largest neg. value:	$-1.7976931348623158 \times 10^{308}$		
smallest neg. value:	$-2.2250738585072014 \times 10^{-308}$		
zero:	0.0		
smallest pos. value:	$2.2250738585072014 \times 10^{-308}$	DBL_MIN	float.h
largest pos. value:	$1.7976931348623158 \times 10^{308}$	DBL_MAX	float.h

channel

number of bits:	32

Random notes concerning the Par.C types

All integer types are signed by default. When (unsigned) shorts or (unsigned) chars are used in expressions, they are first promoted to the corresponding int type. All promotions from signed types to larger signed types are sign preserving.

The integer types differ widely in efficiency. This is due to the transputer hardware, which does not have the addressing modes to support all types directly. We discuss the types below.

char

The transputer character is basically unsigned. Extra code is needed to extend a signed character to an integer. In routines which manipulate strings, it often pays to use unsigned char in stead of the default signed char.

short

There is no addressing mode for short integers. Therefore, the byte addressable block move of the transputer is used to handle them. This means that any access to a short object takes large amounts of time and code. Signed shorts take even more code than unsigned shorts, because of the necessary sign extension. Probably the only useful applications for shorts are large arrays to save memory and interfacing to alien machines with 16 bit word length.

int

Most operations on signed integers have a corresponding transputer instruction, and therefore signed integers are very efficient. An exception to this is a shift right on a signed integer. This is implemented as an arithmetic shift right, preserving the proper sign of the value. For this preservation extra code is needed.

long

Long arithmetic is partially supported in the transputer hardware. Inline code is used whenever it takes no more than about 16 instructions. More specifically, all bitwise and additive operations are expanded to inline code, whereas multiplication, division and remainder are implemented through function calls. Code generated for long expressions tends to be large. There is little difference in performance between signed and unsigned longs.

Note: .... Many existing programs use the type long for a 32 bit value. Because this range can be handled by int much more efficiently, it is recommended to substitute int for long whenever possible. The type specifiers can be replaced by the following preprocessor command:

```
#define long int
```

Care must be taken in format strings for functions like `printf()`, where the format `%ld` should be replaced by `%d`, and in some runtime functions which require a `long` as parameter. If protability is more important than efficiency `printf("%lx".(long) val);` is recommended.

float

This floating point type conforms to the IEEE single length format. A short description of the IEEE formats can be found in an additional section of this chapter. On the T800, this format is directly supported by the hardware. On the T414 there are some supporting instructions for rounding and packing, which are used in an emulation package.

Note .... Float values are not automatically converted to double when used in expressions. However, if a float is passed to a function as a parameter, conversion to double does take place. Conversely, if a formal parameter of a function is declared as float, this declaration is silently replaced by a declaration of type double.

double

This floating point type conforms to the IEEE double length format. The T800 supports this format directly. On the T414 an equivalent emulation package is used.

channel

The channel data type is implemented by the transputer channel. Auto, extern and static channels are reset by the system. Channels which are dynamically allocated must be reset by the user, using the runtime function `ResetChannel()`.

Word alignment of data

As a rule, every object is aligned on a word (4 byte) boundary, even the single components of a structure.

There are two exceptions: Arrays of characters and arrays of shorts.

In arrays of characters and in arrays of shorts, the elements are packed, occupying consecutive memory locations. Note however, that single variables of type char and short take up a full word of storage.

Example ....

```
int u;              u         ****
struct {
   int a;           v.a       ****
   char b;          v.b       *---
   short c[3];      v.c       **** **--
   int d;           v.d       ****
   char e[5];       v.e       **** *---
} v;
char w;             w         *---
long x;             x         **** ****
```

In this example every asterix (*) represents one byte of storage, and every dash (-) represents an unused byte, v represents a record name, but does not specify any memory usage, which is done by the field names a, b, c, d and e; u, w and x are variables; c[3] means an array of three shorts and e[5] means an array of five characters

The byte and word order in memory is always from low to high:

Example ....

```
union{
   long ln;
   struct {
       int lsw,msw;
   } i;
   char c[8];
} overlay;
```

This union defines the following equivalence:

```
physical byte offset   01234567
           overlay.ln  ********
           overlay.i   lsw msw
           overlay.c   01234567
```

`i.lsw` holds the least significant word of `ln`;

`c[0]` holds the least significant byte of `ln`;

`c[3]` holds the most significant byte of `i.lsw`.

Floating point formats

The T800 implements the single and double length floating point formats as defined by the IEEE 754 standard (IEEE [1985]). Of course, this implementation of floating point numbers is embedded in the Par.C System. A short description follows.

The letters in the following description have the meaning:

s	: sign
e	: exponent
m	: mantisse
q	: exponent bias
b	: number of bits in mantisse

The bytes and bits are shown from most significant to least significant.

float

$q = 127 \quad b = 23$

```
seeeeeee  emmmmmmm  mmmmmmmm  mmmmmmmm
31     24 23     16 15     8  7      0
```

double

$q = 1023 \quad b = 52$

```
seeeeeee  eeeemmmm  mmmmmmmm  .. .. .. ..  mmmmmmmm
63     56 55     48 47                            0
```

If you interpret the bitfields s, e and m as unsigned values, the represented value can be calculated with:

$$v = (-1)^s * 2^{e-q} * (1 + m * 2^{-b})$$

where q and b are the values given with the formats above.

Example: ....

float	3f800000 (hex) = 1.0
double	c018000000000000 (hex) = -6.0

Rounding

In most floating point operations, the result is rounded to the nearest representable value, the only exception being conversion from floating types to integral types. In this case the fractional part is dropped.

Glossary

In the following list, the most important terms used throughout this manual are explained. Technical terms related to the functioning of the transputer are not included. For more information on these terms we refer to the transputer reference manual [Inmos, 1986].

Active process

Process which is either running or ready to run.

Active process list

Linked list of all processes ready to run, i.e. all processes excluding the running process and waiting processes.

Concurrency

More than one process sharing the same processor.

Descheduling

Interrupting execution of a running process in order to schedule another active process.

Network

A set of interconnected processors.

Parallel C

The superset of standard C that is recognised by the Par.C Parallel C compiler and supported in the Par.C System.

Process

A list of instructions which are executed sequentially while using a private workspace.

Re-entrant

A function is re-entrant if it can be called from more than one concurrent process at the same time without any conflicts. In practice this means that any state variables should be local to the process or read-only.

Replicator

A series of expressions, used to repeat a piece of code a number of times. The variables used in the replicator are called the replicator variables.

Running process

Process of which instructions are currently being executed.

Scheduling

Taking a process from the active process list and resume its execution.

Task (running)

A set of one or more interrelated processes, which all share a global data space.

Terminated process

Process which has been active, but will not be activated anymore; the workspace is deallocated.

Timeslice

The maximum amount of time a process is allowed to run before it can be descheduled.

Transputer

An Inmos microprocessor of the transputer type.

Waiting process

Process which is waiting for an event to resume execution. The events can be the completion of a communication or a specified timervalue being reached.

Installation

The Par.C System package

Introduction

The Par.C system is delivered on several floppy disks. For the contents of these disks read the readme file on disk 1. This file also contains important, last minute information about the Par.C System which we could not yet include in this manual.

The rest of this chapter will describe what the minimum requirements are for the Par.C System and how to install it. It will be concluded with a simple test to see if the installation went right.

Par.C System requirements

The miminum requirements for the native version of the Par.C System are one transputer (of type T4XX or T8XX) with 256 KBytes of RAM. To be able to produce larger programs, either a larger amount of memory is needed, or the program should be split up into a number of separately compiled modules. With the Par.C System, programs can be made which run on transputers with no external memory.

Where the target transputer network is concerned, transputer boards from different producers are supported, insofar as the transputer network can be booted up through a linkconnection with the host system. This implies that all transputers in the network must be reset before the program is loaded.

If you have any problems with specific transputer hardware products please contact your dealer or Parsec Developments.

Installation

The installation procedure is the same for systems using an IBM PC (or compatible) as host and those that use the Helios Operating System, provided that the Helios Operating System uses an IBM PC as file server.

The Par.C System comes with an installation program, `install.exe`, which copies the system from the source disks to the drive and sub-directory indicated by the user, which will be taken as the "root" directory for the Par.C System. This parc directory can be a subdirectory of any level and will be indicated in this manual by `parc\`.

`Install.exe` sets up the system in the standard way depicted below. The user may choose a different configuration, but the `inc\` and `lib\` sub-directories will have to be present in `parc\`, since the names `inc` and `lib` are hardcoded in the search path used by compiler, linker and loader.

A standard configuration of the Par.C System on disk should look like this:

```
parc\

   hardware.cnf          (hardware info for loader/server)
   bin\
       parcc.run/exe     (compiler and preprocessor)
       parca.run/exe     (assembler)
       parcl.run/exe     (linker)
       run.exe           (loader/server)
       parc.bat          (batch file)
   lib\
       *.lib             (libraries)
       *.rsl             (resident libraries)
   inc\
       *.h           (declarations of library functions)
   util\
       *.exe         (utilities)
   examples\
```

The PARSEC environment variable should be set to hold the name of the "root" directory of the Par.C System. An example is given below:

```
set PARSEC=c:\transp\par.c\         (PC or)
set PARSEC=/c/helios/transp/par.c/ (Helios)
```

The compiler will then search for include files using the subdirectory name formed by appending `inc\` or `inc/` to this PARSEC variable. The libraries will be searched by the linker in a similar way.

Adjustment to interfacing hardware on PC

The Par.C loader/server is designed to work with a number of different transputer hardware systems. The default link-interface used by the loader/server is configured according to the Inmos standard on their IBM-PC interface (as on the B004 plug-in board). The default linkadaptor base address can be overruled using the `-p` option of `run.exe`.

When using hardware with different reset modes, link-adaptor base address and link-adaptor address offset, the `hardware.cnf` file should be changed accordingly.

The `hardware.cnf` file describes the hardware used as interface between the host and the Transputer network. It typically contains three lines which inform the loader/server about the address of the interface adaptor, the link-interface to be used (if the interface adapter is equiped with more than one link-interface) and the way the Transputer network should be reset (which depends on the manufacturer of the Transputer system).

- The adaptor base address is set by the line `Adaptor-base xxx`. Here xxx is the hexadecimal value of the base port address of the link adaptor.
- The number of the link interface is set by the line `Adaptor N`. Here N is a decimal number indicating the link interface. If only one link interface is available N should be zero, or this line can be omitted entirely.
- The reset mode is selected by the line `Reset-mode NAME`. Currently, only the Inmos and Parsytec reset modes are supported. Hence, NAME should be "Inmos" or "Parsytec".

If this file is not available the loader/server defaults to the following settings:

```
Adaptor-base 150
Adaptor        0
Reset-mode Inmos
```

Adjustment to interfacing hardware for Helios

The Par.C loader/server is designed to work with a number of different transputer hardware systems. The default link, used to access the subnetwork where Par.C executes, can be overruled using the `-p` option of `run.exe`.

When using hardware with different reset modes or linkadaptors, the `hardware.cnf` file should be changed accordingly.

The `hardware.cnf` file describes the hardware used to interface the host to the Transputer network. It typically contains two lines which inform the loader/server about the number of the link to use, and the way the Transputer network should be reset (which depends on the manufacturer of the Transputer system).

- The number of the link is set by the line `Adaptor N`. Here `N` is a decimal number indicating the link interface (0 to 3).
- The reset mode is selected by the line `Reset-mode NAME`. At the moment only the Inmos and Parsytec reset modes are supported. Hence, `NAME` should be "Inmos" or "Parsytec".

If this file is not available the loader/server defaults to the following settings:

```
Adaptor    2
Reset-mode Inmos
```

Confidence testing

After installing the Par.C System on the host computer and adjusting parc.bat to the chosen configuration, compiling, assembling and linking a program can be done by typing:

```
parc <filename>
```

If compilation, assembly and linkage of the program have been succesful, an executable program is produced which can be loaded into the transputer network by typing:

```
run <filename> [arguments]
```

If the hardware interface information contained in hardware.cnf is correct, running the example program dynsel.run should give a result similar to the output shown in the example below. Otherwise, check on the availability of hardware.cnf (the loader expects the file to be placed in the parc\ root directory but will also search the path) and check on the contents of hardware.cnf. The source code for the program dynsel.run can be found in the file dynsel.c in the examples directory.

Example .... Possible output of `dynsel.run`

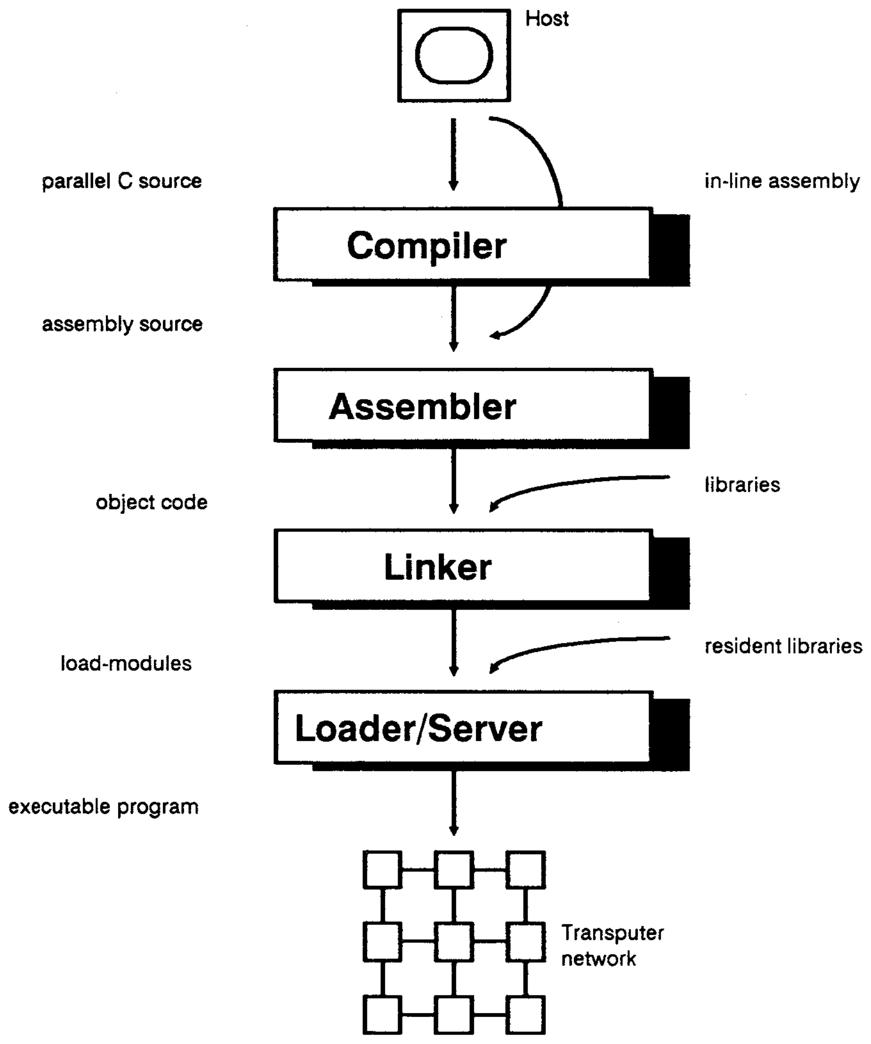
```
Time Out in Select
Sent 25 over channel at 803fcedc
Received 25 from channel C[5] at 803fcedc
Time Out in Select
Sent 0 over channel at 803fcec8
Received 0 from channel C[0] at 803fcec8
Time Out in Select
Sent 1 over channel at 803fcecc
Received 1 from channel C[1] at 803fcecc
Time Out in Select
Time Out in Select
Sent 4 over channel at 803fced0
Received 4 from channel C[2] at 803fced0
Sent 16 over channel at 803fced8
Received 16 from channel C[4] at 803fced8
Time Out in Select
Sent 9 over channel at 803fced4
Replicated par terminated
Received 9 from channel C[3] at 803fced4
All data have been received
Selecting process terminating
```

Because `dynsel` creates a number of concurrent (parallel) processes the actual results can differ from run to run.

More examples are given in the remainder of this manual. It is advised to first read the chapter on the parallel C language extensions.

Par.C System

Overview of the Par.C System



Overview of the Par.C System

The preprocessor is integrated with the compiler, and recognises all Draft Proposed ANSI C preprocessor directives, as well as some compiler-specific directives, e.g. for optimization of code for floating point transputers and for generating non-interruptable code.

The compiler recognises the Kernighan & Ritchie C definition, as well as the extensions that have been added to permit the development of parallel programs. The output is an assembly source file. Inline assembly can be used anywhere in the C source code.

The assembler translates assembly source files into transputer object code, resolving all references internal to a file and optimizing for the length of references, constants and instructions.

The linker has a number of options, making it possible to optimise the performance of the program, e.g. by placing the data area, the code and/or the stack in the fast on-chip memory of the transputer.

The loader/server loads the program into the transputer network and serves the I/O requests of the program. A program can be loaded into any network, regardless of configuration and size.

The runtime system provides support for running parallel programs on a network of transputers, and for dealing with concurrent processes on the same transputer accessing the server when calling I/O routines. The most important parts of the runtime system are explained in the chapter "the runtime system".

The runtime libraries provide standard C facilities, as well as low-level functions supporting specific features of the transputer. Functions which have been defined in the Draft Proposed ANSI C standard in a non re-entrant way have been given a re-entrant alternative.

Compatibility with Draft proposed ANSI C

The Par.C compiler recognises the Kernighan & Ritchie [Kernighan 1978] C standard, with some of the extensions defined by Harbison and Steele [Harbison 1984]. The Draft Proposed ANSI C preprocessor directives are also included [ANSI 1986].

The Par.C runtime libraries are compatible to the Draft Proposed ANSI C standard, with the exception of a small number of functions. These are listed in the Library Reference.

Compiler & Preprocessor

Usage of the compiler

Compilation of a parallel C program is achieved by calling the compiler with optional switches and the name of the sourcefile to be translated into transputer assembly source code. When no extension is specified, ".c" is taken by default. When invoking the compiler without specifying a sourcefile or options, the usage is printed on the screen.

The options which will then be shown are divided into a number of classes. More options in the same class can be invoked by concatenating syllables following the option class specifier. For example, the option "-tsi" will render the same results as using "-ts" and "-ti" seperately.

Most options are obvious in their use. Remarks on some of the special options are given in the following paragraphs.

Input- and output file names

Options ....

```
parcc -o<file> <inputfilename>  /* outputfile */
parcc -i<dir> <inputfilename>   /* include directory */
```

The name of the outputfile can be specified using the "-o" option. The default outputfilename is the inputfilename with a ".s" suffix. The specified outputfilename is used exactly as it is given. The default extension for the inputfilename is ".c".

Example: ....

```
parcc -omyfile testprog
```

This will cause the file "myfile" to be used as outputfile for the compilation of "testprog.c".

Using the "-i" option enables the use of include files from other directories than the default include directory. The compiler will search for include files in the following directories in the indicated order:

- The user defined directory following the "-i" option.
- The directory formed by appending "inc\" to the PARSEC environment variable.
- The default include directory "parc\inc\".

Example: ....
```
parcc -id:\mydir
```

This will cause the compiler to use the directory "mydir" on drive D: to search for include files. Included files that cannot be found there will be searched in the directory defined in the PARSEC environment variable. If this variable is missing, or if the file searched for cannot be found there, the directory "\parc\inc" on the current drive is searched. If this does not lead to the include file being found and used, the compiler will exit with a fatal error message.

Alignment on word-boundaries

Options ....
```
parcc -af <inputfilename>       /* alignment of functions */
parcc -al <inputfilename>       /* alignment of loops     */
parcc -alf <inputfilename>      /* both alignment forms   */
```

Word-alignment may affect the execution speed of the program. Instructions of the transputer are coded in bytes, but are collected in words (four bytes) because of the word oriented memory interface of the transputer. Since most of the instructions need more bytes to be built up (see the transputer hardware information provided by Inmos [Inmos 1986]), an instruction crossing word-boundaries will take 2 memory accesses to be read into the transputer's instruction register. If a loop is executed for a large number of times, alignment of the loop can give some increase of performance. The same goes for functions which are executed very often in the course of the program.

Example: ....
```
parcc -alf dhry
```

This will cause the compiler to emit code with word-aligned functions and loops. To be effective, the program should not be linked with the "-a" option of the linker, because in that case the start of the module is not necessarily aligned.

The effectiveness of word alignment is limited. Especially when the floating point unit is being used, the speed of execution depends strongly on the timing of the memory accesses required by the fpu. Memory access by the link engines may also cause different timings.

Using an indirect file

Options ....
```
parcc @<indirect file>
```

The Par.C Compiler can be instructed to take its arguments from an indirect file. This can be done by preceding the name of the indirect file with '@' on the commandline. The indirect option can be mixed with other options and filenames. It behaves like replacing the indirect file specifier with the contents of the file, except that newlines in the file are treated as whitespace.

Example: ....

```
parcc -ts @std_c_opt myprog.c -o test.s
```

This command will compile the file `myprog.c` to the outputfile `test.s`. The compiler will include the lines from the source file as comment in the assembler file, and use additional options from the file `std_c_opt`.

If the file `std_c_opt` contains the line `-ti -i\myincdir` the compiler expands this into `-ts -ti -i\myincdir myprog.c -o test.s`

Note: .... Indirect files can not be nested.

Warning levels

Options ....

```
parcc -we <inputfilename>       /* exit code on warnings */
parcc -wl <inputfilename>       /* set warning level     */
```

The "-we" option may be useful when the compiler is called from within another program, which can then use the special warning exitcode to act appropriately (see the last section of this chapter "exitcodes").

The "-wl" option makes it possible to indicate the level of warnings which should be suppressed. The option should be followed by a single decimal digit between 0 and 6 inclusive. Setting the warning suppression level to 0 will cause all warnings to be displayed.

Preprocessor directives

The following standard C preprocessor directives are supported:

```
#define                    #undef
#ifdef                     #ifndef
#if                        #else
#endif                     #include
#asm                       #endasm
#line                      #elif
#pragma
```

The last three directives mentioned above are included in the Draft Proposed ANSI C standard, and have the following meaning:

```
#line <number>
```

sets the line-counter to the number specified.

`#elif`

is used as contraction of `#else` and `#if` appearing on consecutive lines.

`#pragma`

precedes compiler-dependent directives, which are skipped when another compiler is used which does not recognize them.

Compiler-dependent directives

The compiler dependent `#pragma` directives supported in Par.C can be divided in directives related to the use of a transputer containing a floating-point unit and generation of non-interruptable code.

Inline floating point code generation

Usage ....

```
#pragma fpu
#pragma nofpu
```

When compiling statements using floating point calculations, the generated code is generic for transputers with or without floating point unit. In combination with this, the Par.C Runtime System enables the use of the same code on different transputers, because different libraries containing floating point primitives are loaded on different transputers and linked during loadtime with the program code.

In most cases, the use of function calls for all floating point primitives will cause a considerable overhead when running the program on a transputer with floating point unit. Using the `#pragma fpu` directive will cause the compiler to generate floating point instructions directly. After `#pragma nofpu` the generated code will be generic again, so that parts of the program can be made dependent on the available transputer types.

Example: .... Use of the `#pragma fpu` directive

```
#pragma fpu
void DoFloats();
{
   ...
}

#pragma nofpu
void PassMessages();
{
   ...
}
```

```
main()
{
   par
   {
      if(_ttype==8) DoFloats();
      PassMessages();
   }
}
```

In this example, a structure is given for a program which should run on any transputer network, containing a number of T8XXs which are interconnected using T4XXs. The latter processor type will only have to pass messages from the host system to the T8XXs and back. Since `DoFloats()` is only started on T8XXs, the program as a whole will run on any transputer network. Note that the code for the floating point calculations will also be loaded into the T4XXs, but will never be executed there.

Note . . . . The program will hang if code compiled with `#pragma fpu` is run on a T4XX.

Uninterruptable code generation

Usage . . . .

```
#pragma noint
#pragma int
```

Some of the instructions of the transputer are defined as descheduling points. This implies that a process running on low priority may be descheduled at these points. Most of the descheduling points concern waiting for communication or a timer, but descheduling is also possible when a process has been running for more than two timeslice periods (of 1024 microseconds) each. To prevent execution of certain pieces of code from being interrupted, the directive `#pragma noint` can be used. The compiler will in this case generate only non-interruptable instructions. Using `#pragma int` will have the compiler return to generation of normal code.

Note . . . . Code for the execution of statements having to do with communication or a timer will still use instructions at which the process can be descheduled. A full list of interruptable instructions can be found in the Transputer Instruction Set [Inmos 1986].

Predefined macros

The predefined macros defined in the Draft Proposed ANSI C standard are supported in Par.C. These are the following:

`__LINE__`	replaced by the current linenumber in the sourcefile
`__FILE__`	replaced by the current sourcefile name
`__DATE__`	replaced by the current date
`__TIME__`	replaced by the current time
`__STDC__`	replaced by FALSE in this Par.C compiler

Aside from these, the following macros are predefined in Par.C and will probably not be supported in future versions:

`__FUNC__`	replaced by the current function name
`__RAND__`	replaced by a random generated number

In order to recognise the Par.C compiler a macro is defined:

`__PARC__`	defined if Par.C is used, replaced by 1

Error recovery

The type of error recovery used in the Par.C compiler is the so-called "panic-mode". When the compiler encounters an error, characters are simply skipped until some synchronising character is reached (for example a semicolon at the end of an expression statement, or a right brace at the end of a compound statement). This mechanism is not very sophisticated. Quite often too much text is skipped, or compilation is resumed at the wrong position in the source text. When this happens, an avalanche of error messages will be displayed, most of which will be caused by the compiler skipping some text (for example a declaration).

In most cases, only the first few error messages are significant. Missing or superfluous semicolons are infamous for causing insignificant error messages.

Exitcodes

The compiler exits to the calling program with an exit code, specifying the status of what has been compiled. The exit code may have the following values:

exit code 0 : No errors. No warnings. Compilation successful.

exit code 1 : Warnings but no errors. Compilation successful.

exit code 2 : Errors in the source. Compilation continued.

exit code 3 : Fatal errors. Compilation aborted.

exit code 4 : System errors. Compilation aborted.

The difference between exit codes 0 and 1 is only in effect when compiling with the "-we" option. When this option is not switched on, the compiler will not differentiate between levels 0 and 1, and in both cases return 0.

Exit code 2 is generated when errors have been found in the source code, which have not led to abortion of compilation. Exit code 3 is generated when fatal errors have been encountered, either in the command line or in the source code.

System errors are for example insufficient memory or internal errors in the compiler.

Assembler

Usage of the assembler

Assembling a transputer assembly source file (with suffix ".s" by default) is achieved by invoking the assembler with options and the name of the file to be assembled into transputer object code. When invoking the assember without specifying a sourcefile or options, the usage is printed on the screen.

Outputfile name and format

Options ....

```
parca <inputfile> -h           /*hex output instead of bin*/
parca <inputfile> -ic<number>  /*specify length code label*/
parca <inputfile> -id<number>  /*specify length data label*/
```

The output file can be written in hexadecimal notation using the "-h" option on the commandline.

Example: ....

```
parca hello -h
```

This will cause the assembler to write the output file named "hello.o" in hexadecimal notation.

The length of external code and data references can be specified with the "-ic" and "-id" options. These options can be used to optimize code when no large reference offsets will be needed in the program.

Example: ....

```
parca dhry -ic 3 -id 4
```

This will cause the assembler to reserve three bytes for each unresolved code reference and four bytes for each unresolved data reference.

Memory usage of the assembler

Options ....

```
parca <inputfile> -sc <number> /*size of code table      */
parca <inputfile> -se <number> /*size of expression table*/
parca <inputfile> -sl <number> /*size of symbol table    */
```

The assembler is optimized for speed and as a consequence uses a lot of memory to run. This may sometimes cause problems for larger programs, when one of the used tables is not large enough to contain all information needed. Since the assembler allocates all available memory when starting and divides this over the tables, adjusting table sizes may solve the memory problems. The size of the non-specified tables are not adjusted automatically when changing the sizes of one or two of the tables.

The number following each of the "-s" options should be given as a decimal number indicating the number of elements to be reserved in the array.

Example: ....

```
parca hugeprog -sc 5000 -se 2500 -sl 500
```

This will cause the assembler to declare an array of 5000 elements for the code table, 2500 elements for the expression table and 500 for the symbol table. The default settings of code, expression and symbol array sizes have ratios 4 to 4 to 1. The element sizes are 32, 64 and 8 bytes long. When running into memory problems and trying to find out how much elements to use, running the assembler with the "-a" option may give some indication, since the number of elements used in each of the tables will be displayed.

If the use of these switches does not solve the memory shortage, there are two possibillities: Either split the source file in smaller parts, or buy more memory.

Listfile name and settings

Options ....

```
parca <inputfile> -la          /*write all to list file  */
parca <inputfile> -l           /*write according to .list*/
parca <inputfile> -lf<file>    /*specify list filename   */
```

The listfile contains all assembly source code, all object code (in hex format) and all labels. When assembling in the default mode, the list file is not generated. When using the "-la" option, all source is written to the list file. The "-l" option will cause only assembly source which is not preceded by a ".nolist" directive to be written to the list file. The default list filename "t.l" can be changed using the "-lf" option.

Using an indirect file

The Par.C Assembler can be instructed to take its arguments from an indirect file. This can be done by preceding the name of the indirect file with '@' on the commandline. The indirect option can be mixed with other options and filenames. It behaves like replacing the indirect file specifier with the contents of the file, except that newlines in the file are treated as whitespace.

Example: .... `parca myprog @stdasopt`

This command will assemble the file `myprog.s` with standard assembler options taken from the file `stdasopt`.

Note: .... Indirect files can not be nested.

Linker

Usage of the linker

Linking a number of transputer object code files and libraries to an executable program is achieved by calling the Par.C linker with options. When invoked without arguments the usage of the linker will be displayed on the screen.

Files and options are allowed to be mixed on the commandline, provided that an option is always preceded by a dash '-' (with the exception of the indirection option "@<filename>"). The commandline options that are recognized by the linker are explained in the following paragraphs.

The linker assumes the following suffixes for files which are named without suffix:

`.o`	for object files
`.lib`	for library files
`.run`	for executables
`.map`	for map files

Outputfile name and format

Options ....

```
parcl -o <inputfile> /* specify output filename   */
parcl -h <inputfile> /* hex output instead of bin */
```

The "-o" options allows you to specify a filename for the linker output. If this option is omitted, the linker will use the name of the first object file in the list, with suffix ".run". The "-h" option will cause the linker to create a file with a hexadecimal representation of the executable.

Example: ....

```
parcl hello -odemo
```

This will cause the linker to create the executable file "demo". Note that no ".run" extension will be appended in this case.

Specification of library files

Option ....

```
parcl <inputfile1> <inputfile2> ... -l<lib1> -l<lib2>
```

The "-l" option is used to indicate which libraries are to be searched during the process of linking. Unless told otherwise, the linker will search for its libraries in the current directory, and if not found there, in the subdirectory `\lib` of the directory that has been specified by the environment variable PARSEC. See the "Installation" chapter on the use of this variable in the Par.C System. A full pathname is allowed after the "-l" switch. Modules from files specified with the "-l" option will only be linked to the executable file if they could not be resolved from object files.

Example: ....

```
parcl hello -la -lio -lstd -ld:\mydir\mylib
```

This will cause the linker to use the libraries `a.lib`, `io.lib` and `std.lib` in the default library directory, and `mylib.lib` in the directory `\mydir` on drive `D:`

Generation and format of the mapfile

Options ....

```
parcl <inputfile> -m        /* set generation of mapfile  */
parcl <inputfile> -ml       /* include a library map      */
parcl <inputfile> -mf<file>/* specify mapfile name       */
parcl <inputfile> -mx       /* add cross reference to map */
```

The "-m" options causes the linker to write a map file. By default this file will have the name of the first object module in the list, with suffix `.map`. After the "-mf" option, you may add a filename for the mapfile.

The "-ml" option causes the linker to write a map file which includes a map of the libraries. The "-mx" option causes the linker to add a cross reference to the map file.

Example: ....

```
parcl demo -mxlf mymap.ttt
```

This will cause the linker to generate a mapfile with name `mymap.ttt` which includes a map of the used libraries and cross references.

Specification of the executable program

```
parcl <inputfile> -bc       /* place code in low memory  */
parcl <inputfile> -bd       /* place data in low memory  */
parcl <inputfile> -bs       /* place stack in low memory */
parcl <inputfile> -s<size> /* set stacksize to <size>   */
parcl <inputfile> -c        /* do not link a booter      */
```

The "-bc" option will cause the code segment to be loaded as low as possible in memory. Since the lowest 2 or 4 Kbytes of the transputer are on-chip memory (4 Kbytes on a T8xx), this memory has a very low access time. In this way the execution speed of a program may be increased.

The "-bd" option will cause the data segment to be loaded as low as possible in memory. In this case it is not the execution of the code that will be speeded up, but the access to static and global variables.

The "-bs" option will cause the initial stack to be allocated in low memory. This will speed up the access to local variables.

The "-s" option allows you to specify an initial stacksize for your application. The "-s" accepts an argument which determines the stacksize to be used. If this argument is preceded by '0x', it is assumed to be a hexadecimal number. If it is preceded by '0', it is assumed to be an octal number. Otherwise it is interpreted as a decimal number. There is no maximum to the size of the stack, other than the amount of available memory. The linker will use a minimum stacksize of 64 words. At runtime the stack will be expanded if necessary. (See the description of the runtime function `_setFunstack()` for more details.)

Example : . . . .

```
parcl <inputfile> -bc -s0x100
```

This will cause the linker to create an executable program which will have its code segment loaded in low memory and which uses stacks of 256 (0x100) bytes.

The "-c" option will cause the linker to refrain from including booter code. Booter code is among other things used to distribute an application over the transputers in the network. When creating a program that is not meant to be loaded into an unbooted transputer network, this option should be used. In this case it is the programmer's responsibility to load the code and start its execution. For example the resident libraries of the Par.C System ("`realt4.rsl`" and "`realt8.rsl`") have been created this way.

Using an indirect file

The Par.C Assembler can be instructed to take its arguments from an indirect file. This can be done by preceding the name of the indirect file with '@' on the commandline. The indirect option can be mixed with other options and filenames. It behaves like replacing the indirect file specifier with the contents of the file, except that newlines in the file are treated as whitespace.

This option can save a lot of typing and it can be used to circumvent restrictions on the size of the commandline.

Example: .... `parcl myprog -la -lio -lstd @myobjects`

This command will link the file `myprog.o` with several libraries and will read `-l` options from the file `myobjects`, where the user has listed his other modules.

Note: .... Indirect files can not be nested.

Loader/Server

Usage of the loader/server

The Par.C loader/server (run.exe) has a dual function. First it loads a program into the transputer system, and gives control to the transputer. Secondly, the loader/server will serve any I/O requests the program might issue to the host computer. When invoked without arguments, the usage is printed on the screen.

All the options that are typed before the name of the program are interpreted as options for the loader. The first filename is regarded as the name of the program to run. The arguments following the program name are passed to the program as commandline arguments.

When the loader is instructed to load a program, it first searches for it in the current directory. If no match is found, it will search the subdirectory indicated by the PARSEC environment variable. When the program is not found in the "parc\" root directory, the subdirectory "PARSEC\bin\" is searched. Following this, the MS-DOS path is searched. If the program is still not found, the loader will exit after printing an error message.

When needed for the execution of the program, the resident libraries `realt4.rsl` and/or `realt8.rsl` are loaded. These libraries contain the floating point primitives for T4XX and T8XX transputers and are only loaded if the program uses floating point calculations. The loader expects the libraries in the subdirectory "PARSEC\lib\" where the other libraries are also located.

Commandline options

Option .... `run -i <program> [arguments]`

The "-i" option will cause the first transputer's memory to be cleared before the program is loaded into it. This can be useful when testing a program and using post-mortem debugging tools to investigate variables in the transputer's memory.

Option .... `run -p<address>  <program> [arguments]`

The "-p" option will overrule the default link adapter port address. This option provides a way to have more than one link connection with one or more transputers. It also offers an easy way of experimenting with other link adapters without having to edit the file "`hardware.cnf`".

Option .... `run -r <program> [arguments]`

The "-r" option will keep the server from resetting the transputer network before loading a program into it. This may be useful in cases where a program is already active which should not be killed when another program is loaded.

Option .... `run -s <program> [arguments]`

The "-s" option will give a detailed description of the loader options that have been used on the commandline or the configuration file. It will also print the full pathname of the configuration file that has been used.

Option .... `run -t <program> [arguments]`

The "-t" option will time the execution of the program. Timing will start after the program has been loaded, and it will stop when the program issues an exit request. The elapsed time is then written to the standard output.

Option .... `run -v <program> [arguments]`

The "-v" option will cause the loader to print its own filename and a copyright message to the standard output (the compiler, linker and assembler always do so).

Option .... `run -b <program> [arguments]`

The "-b" option causes the loader to open files in binary mode regardless of the mode that is specified in the `fopen()` call. Omitting this option causes the file to be opened exactly as is specified in the `fopen()` call. This option is useful on MS-DOS systems and is usually provided by MS-DOS C compilers to provide a way to resolve the difference between the UNIX and the MS-DOS method for indicating an end of line. When a file is opened in text mode, every newline character ('\n') is translated to a CR/LF sequence during writes. Vice versa a CR/LF sequence is translated to a newline character during reads.

Option `run -x <program> [arguments]`

The "-x" option will cause the loader to exit immediately after loading the program. Thus the server part will not be started. This way the programmer is able to use the loader as a means to start a program, and then serve transputer requests with a custom server.

Option .... `run -d <program> [arguments]`

The "-d" option causes the loader to display all I/O requests of the transputer program to the screen. The information obtained can be used to trace the progress of the running program.

The runtime system

General description of libraries

To produce an executable Par.C program it has to be linked with one bootlibrary, one I/O-library, the standard utilities library and an optional math library. Except for the standard utilities package, a choice can be made for each of the libraries, according to what is needed in each particular case.

Boot libraries

Using `a.lib` will produce a single-transputer program. Using `b.lib` will produce a multi-transputer program and cause all available transputers to run the program.

The special library `run.lib` must be used when the program uses `Run()` to boot subprograms into subsystems. See the description of `Run()` for more details.

I/O libraries

When the program performs file I/O, `io.lib` should be linked. In other cases `noio.lib` must be linked instead. This will cause the server to exit after the program has been started. Communication can then be performed through a custom server produced by the user.

Note: .... `io.lib` and `noio.lib` on one side and `std.lib` on the other are heavily cross referenced. The only reason for not joining them in one library is to allow for `noio.lib` to be linked instead of `io.lib`. In the interest of code compactness, cross reference offsets have been limited with the use of the assembler "-ic" and "-id" switches. As a result of this, large amounts of code between `io.lib` and `std.lib` can result in an error message from the linker. It is recommended to specify these two libraries directly after each other.

Standard utilities

As mentioned, `std.lib` must always be linked, since it contains all basic support for Par.C programs.

Math libraries

Two math libraries are available: `math.lib`, which is generic for T4xx and T8xx transputers. The routines contained in it make use of the resident libraries `realt4.rsl` and `realt8.rsl` for low-level floating point support.

The T8XX library `t8math.lib` can be linked instead of `math.lib` to increase performance on math routines, when executing the math code on T8XX processors only. The speed improvement on math routines amounts to a factor of 10 on average (compared for T8XX in both cases).

The Par.C file I/O system

The Par.C file I/O system has been redesigned and will in the near future be extended to also support multi-transputer file I/O. The global and detailed descriptions of the previous manual were therefore removed from this edition.

At the moment we support only file functions on the transputer directly connected to the host (T1). Using these functions on other transputers will result in a crash.

Exiting, breaking and aborting programs

The different routines related to termination of a running program are:

`exit(Ex_val)`

- invokes the 'atexit' routines
- closes all files
- makes the host server exit with value `Ex_val`
- does not return to the calling function

`_exit(Ex_val)`

- makes the host server exit with the value `Ex_val`
- does not return to the calling function
- does not close any files

`abort()`

- raises the signal `ABORT_SIG`
- makes the host server issue a serious error (-2) and exit with code -1
- does not return to the calling function
- does not close any files

`_abort()`

- makes the host server issue a serious error (-2) and exit with code -1
- does not return to the calling function
- does not close any files

`release(Ex_val)`

- closes all files
- makes the host server exit with value `Ex_val`
- returns to the calling function

`*BREAK*` handling

- raises the signal `ABORT_SIG`
- closes all files
- makes the host server exit with the value -1

Network information

The Par.C System was designed to enable programmers to make programs which are independent of the transputer network. This means that a well-written program, once compiled, can run on any network, independent of configuration or topology.

One of the tools available to the programmer for this purpose is the System Network Information. This information is local to each transputer and includes for example the transputer identification number, the numbers of its neighbours and the fastest connection with the host. In the current version of the Par.C System, the information is collected when booting the network. The Par.C loader/server investigates the network at hand by means of a network analyser and leaves information behind on each transputer.

The data structure containing the information is defined as the `SYSTEM` structure in `system.h`. The data can be obtained by a call to the `GetSysInfo()` library function. Refer to the Tutorial for more information on this structure.

Some of the values in the `SYSTEM` structure are also available as external variables. They can be used without the need of calling `GetSysInfo()`. Their use is discouraged however, as in future versions they might be removed.

Parallel C

Introduction

Parallel C is a superset of the standard C language, as proposed by Kernighan and Ritchie [Kernighan 1978] and extended by Harbison and Steele [Harbison 1984]. In parallel C, one datatype and two statements have been added, to enable C programmers to write parallel programs.

In this part of the manual, the language extensions are introduced and their syntax is defined. Each of the chapters describes one extension and contains a syntax section, examples of their use in ordinary cases, a detailed description of the functioning of the extension and a section on pitfalls to be avoided.

Basic concepts of Parallel C

Parallel programming involves the division of the available work between a number of available processors, interconnected to form a network of some shape and size. To be able to divide the work between multiple processors, tasks must be identified which can be executed more or less independent of each other.

But parallel programming implies more than just starting concurrent tasks: at certain points during execution of these tasks, they will have to pass data around or synchronise with each other to ensure a correct parallel control flow of the program, yielding correct overall results. Without this inter-task communication, one could hardly speak of one program, being executed in parallel.

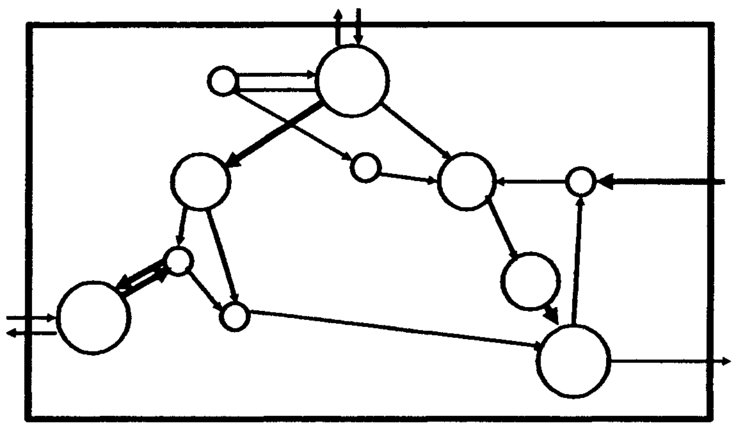
The figure shows the structure of an example parallel program, where tasks are depicted as circles and the arrows show the interconnections used for data transfer or synchronisation.

To be able to execute this example program in a concurrent way, some extra facilities are needed above the possiblities that are available in the standard C programming language.

Available facilities

First of all, multiple processes must be started concurrently, each executing one of the available tasks. To be able to do this in as flexible a way as possible, parallel C offers the par statement, which is described in the next section of this part of the Par.C System Manual.

The second facility that is needed enables data transfer between tasks, and is implemented through the channel datatype. Since neither of the communicating processes can proceed until the datatransfer is completed, the channel also acts as a point of synchronisation. For



Logical map of a parallel program

asynchronous communication see the paragraph "Communication and synchronisation" in the detailed description of the channel datatype.

One more feature is needed, which may be less directly apparent from the parallel structure shown in the figure. Some of the tasks in a parallel program depend on information coming in from different directions, whereas the order in which the data will arrive is not determined before the program actually runs. In this case, a process should be able to wait for a number of events at the same time, resuming execution when any of these occurs. This possibility is provided in parallel C through the select statement.

Language extensions

The choice for language extentions, as opposed to the inclusion of parallel programming facilities in special runtime libraries, has been made for reasons of clarity of concepts and parallel C source code, and also to provide a maximum of flexibility in the use of the parallel facilities.

The extensions to the language have been defined in such a way, that no conflicts arise with the use of specific keywords in standard C. New keywords are only added where they were necessary to implement elementary support for parallel processing in a flexible way.

Reserved keywords in Parallel C

Reserved keywords in parallel C are given in alphabetical order in the table below.

alt(*)	double	int	struct
auto	else	long	switch
break	enum	par(*)	timeout(*)
case	event(*)	process(*)	typedef
channel(*)	extern	register	union
char	float	return	unsigned
cond(*)	for	select(*)	void
continue	goto	short	while
default	guard(*)	sizeof	within(*)
do	if	static	

The keywords marked (*) are specific to Parallel C, and are briefly explained below.

alt	used in select body, to mark one alternative clause
channel	new datatype, used in communication primitives
cond	used in alt clause, to activate or ignore it
event	reserved for future extensions
guard	used in alt clause, preceding a channelpointer
par	new statement, used to start parallel processes
process	reserved for future extensions
select	new statement, used to wait for one of a number of events
timeout	used in alt clause, defining code for timeout
within	used in select header, to define the waiting time

The par statement

Short description

Running several processes concurrently is made possible by the par statement. This statement is used to define parts of the program as separate tasks, which are executed simultaneously by a number of child processes. The parent process, which is temporarily halted, will proceed after all its child processes have terminated.

Example of use

Example ....

```
par {                                                    (1)
   (compound) statement;
   (compound) statement;
       ...
   (compound) statement;
}

par <replicator> {                                       (2)
   (compound) statement;
   (compound) statement;
       ...
   (compound) statement;
}

<replicator> : (initialisation; condition; update)       (3)
```

See also .... The chapter 'Quick reference' for a formal syntax.

A par statement consists of the keyword `par`, followed by the par body enclosed in braces as shown in (1). The par body consists of a number of statements and/or compound statements. For each (compound) statement, a separate child process is started. After termination of all the child processes, the parent process proceeds with the execution of the first statement following the closing bracket of the par statement.

The header of the par statement may include a replicator as shown in (2). The replicator should be of the form shown in (3). The replicated par resembles the loop which is executed in a for statement. However, this replicated par is used to start a number of processes concurrently (instead of having one single process do an iteration). Each of the specified processes inside the par body is then replicated. Details on the replicated par statement can be found in one of the following sections.

The (compound) statements within the par body may be of any kind, according to standard C syntax rules, with the exception of the return and continue statements (the reasons for these exceptions are explained in section "return and continue statements"). This implies also that par statements can be nested.

Use

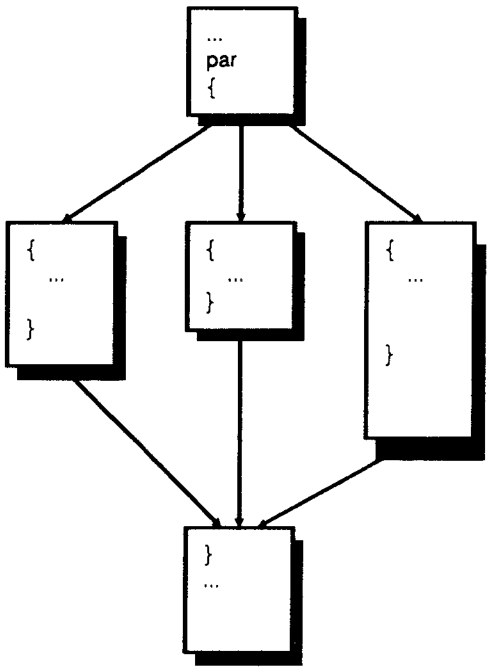
The par statement is used to divide a parent process into a number of tasks, which are executed by separate child processes, running concurrently on the same processor.

The par construct is always contained in the code for one processor, and can therefore not directly cause processes to be started on other processors in the available network. This possibility is handled by calling different tasks or functions (which may each be divided into a number of subtasks) on different processors, depending on the identity of the processor. More information on this can be found in the Tutorial.

The par statement is particularly useful in programs that can be split up into groups of tasks, which are largely independent of each other's proceedings. In the case of strongly interrelated calculations, using the par statement will often turn out to be inefficient.

It is usually efficient to define communication tasks (including file access) as separate processes to prevent having the processor spend too much idle time, waiting for data to arrive over its links. In this way calculating processes can proceed while other processes are waiting for communication to take place (e.g. to transmit results and receive new data and commands for the calculation processes).

The processes inside a par statement are defined as (compound) statements. They may contain function calls, declarations of local variables, conditional expressions, etc. As is noted before, par statements may also be nested. In the example on the next page, some of the possibilities are illustrated.



Execution of par

Example .... Executing five processes in parallel

```
par {
   ProcessOne();                               /* process(1) */
   {                                           /* process(2) */
       int templ, temp2;
       templ = a + b;
       temp2 = c + d;
       e = templ + temp2;
   }
   if (Proc3Valid == TRUE)                     /* process(3) */
       ProcessThree();
   while (TRUE) {                              /* process(4) */
       sleep(1);
       printf(".");
   }
   par {                                       /* process(5) */
       Process5a();
       Process5b();
       Process5c();
   }
}                                          /* end of nested par */
```

Process (1) starts execution with a function call to `Processone()`. After returning from `Processone()`, this first process terminates.

Process (2) contains declarations of local variables and executes a number of statements without performing a function call.

Process (3) uses a conditional statement. If the expression evaluates to FALSE, the process will simply terminate. Of course it is also possible to use an if/else construction here.

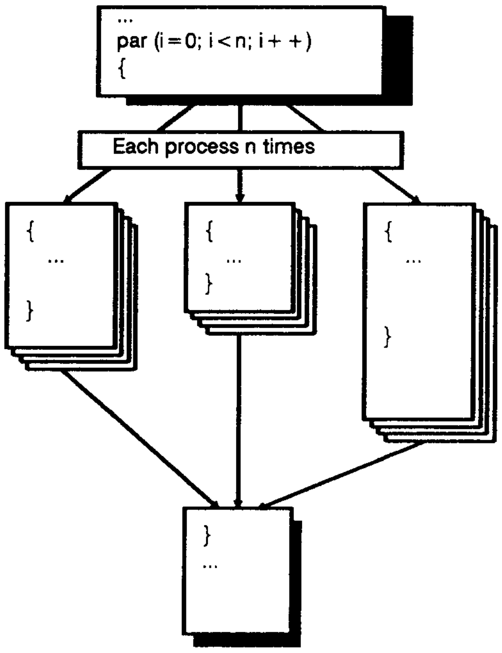
Process (4) uses a `while (TRUE)` construction. Since in this particular case the condition for termination of the while loop is never satisfied, the process will run on indefinitely. As a result, the code following the par statement will never be reached.

Process (5) is a nested par statement. This code causes a process to be started, which in turn executes another par statement. All nested parallel processes must have terminated before process (5) terminates.

Replicated par statement

In the replicated par statement, each of the (compound) statements in the replicated par body is not executed once, but a number of times, which is determined by the replicator following the par keyword. The total number of child processes executing inside the replicated par is therefore the number of (compound) statements times the number of replications.

The replicator included in the header of the replicated par statement closely resembles the replicator used in a for statement, and therefore allows variable boundaries to be used. This implies that the number of times each of the (compound) statements will be started can be determined during runtime. In principle, the variables used in the replicator can be of any type, except for structure or array elements. For details on the reason for these exceptions, see the detailed description of replicated par statements.



Execution of replicated par

The following example illustrates the use of a replicated par statement

Example .... Replicated par statement using an integer

```
par (i = 0; i < n; i++) {
   printf("%c", string[i]);
}
```

Execution of this par statement will cause a number of concurrent processes to be started, of which each one prints one character on the screen.

Example .... Replicated par statement using pointers

```
par (p = q; p != NULL; p = p->next) {
   DoSomethingWith(p);
}
```

In this example the existence of a number of structures containing data to be processed is assumed. A list is maintained by linking each of the structures to the next through a pointer (indicated as `next`). The last structure in the list is marked by giving the next pointer the value NULL. A pointer p can now be used to traverse the list.

In the following example, something is shown which probably does not make sense in any program, but which is still possible as a consequence of using the standard C replicator in combination with the par statement. By using the for statement with an "empty" replicator, iteration will continue indefinitely. Therefore this "for(;;)" is often called the "forever" construct. The same is possible in the replicator of the par statement, yielding an infinite number of child processes to be started.

Example .... The "parever" construct

```
par( ; ; ) {
   DoSomething();
}
```

It should be noted in this case, that, since each child process needs its own newly allocated workspace, available memory will soon run out, and the program containing this "parever" will crash after producing a fatal runtime error message. The `dangerous.c` program contained in the Par.C System package will show this.

Detailed description

In this section, more information is given on a number of topics concerning the par statement. An important special issue in parallel processing is the use of global variables. This topic is addressed, as far as par statements are concerned, in the next section. The special characteristics of the replicator variables in a replicated par statement are clarified in the section "Replicated par statement" below. But first, a notational matter is discussed.

The use of double brackets

In most of the example programs contained in the Par.C System package, double brackets are used in par statements, even where they are not necessary. This is done for reasons of clarity of the parallel C source code. Consider the following example:

Example .... par with single statements

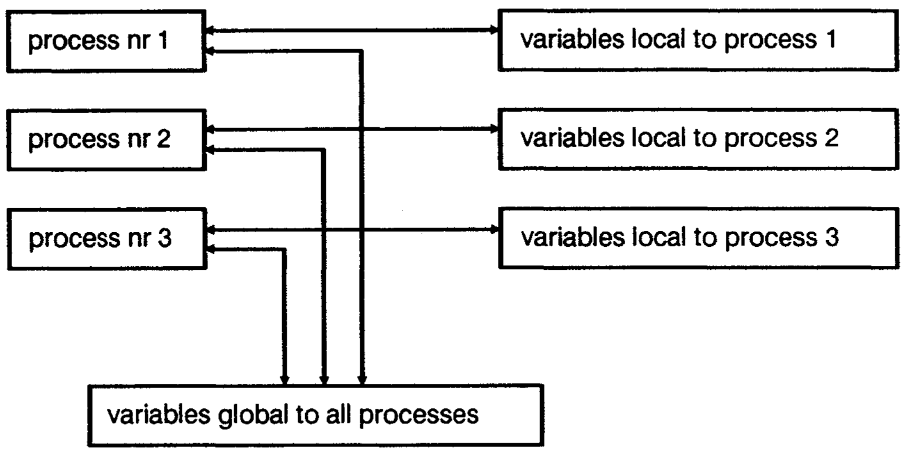
```
par {
    statement1;
    statement2;
    statement3;
    statement4;
    statement5;
}
```

If you want to have statements 1 and 2, and statements 3 through 5 executed in sequence by one single child process each, you have to use extra sets of brackets to make them into compound statements, as is shown in the listing below. On the left side, the extra brackets have been added in a standard C style. On the right side, the placement of these brackets has been slightly adjusted, yielding the "double brackets" that are used in many of the example programs contained in the Par.C System package. This notation may look a little strange, but it helps to keep the programmer from making errors by omitting or misplacing a closing or opening bracket.

Example .... Standard style brackets   Par.C style: double brackets

```
par {                                 par
    {                                 {{
          statement1;                       statement1;
          statement2;                       statement2;
    }                                 }{
    {                                       statement3;
          statement3;                       statement4;
          statement4;                       statement5;
          statement5;                 }}
    }
}
```

The double brackets are also used in a replicated par statement, where only one process should be replicated, containing multiple statements.


process nr 1
process nr 2
process nr 3
variables local to process 1
variables local to process 2
variables local to process 3
variables global to all processes

Use of local and global variables

Using shared variables in concurrent processes

All processes started inside a par statement have access to variables declared in the function containing the par statement (and of course to variables declared outside functions). The same scope rules apply as for other compound statements, except for the variables mentioned in the replicator expressions. For these variables local copies are generated by the compiler, which effects in making the original variables read-only as far as the started processes are concerned. More information on the replicated par statement is given in the next section.

Problems

Using global variables in standard (sequential) C does not cause problems because the order of execution of the program is entirely determined by the structure of the program and the value of variables. No conflicts arise from using and updating variables since the program consists of only one process. Unless hardware failures occur, there can be no changes in variables between the termination of one instruction and the start of the next one.

When dealing with concurrent processes, another factor comes into play, which makes the control flow of the program more complex. The order in which the concurrent processes on the same transputer are executed, not only depends on the structure of the program and the value of variables, but also on time.

The execution of a par statement is illustrated in the previous figures. The process containing the par statement starts a number of concurrent processes, and proceeds to the synchronisation point to wait for termination of all concurrently running processes.

If concurrent processes are using the same variables, problems may arise from the uncertainty on the order of execution of the concurrent processes. There is no guarantee that the used shared variable will keep the same value in the time between descheduling and scheduling of a concurrently running process. Another process with access to the same variables may have changed them in the meanwhile.

To avoid errors, the programmer should take care when using variables global to concurrent processes. Most problems can be avoided by defining local copies of global variables inside the started processes, or permitting only one process to alter the variable.

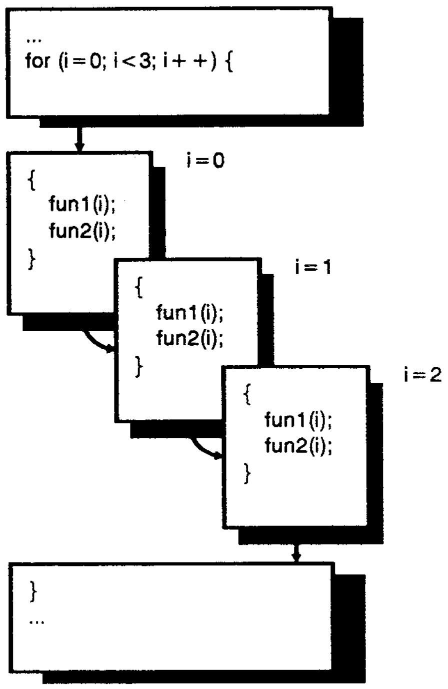
Application

On the other hand, the problems can become a useful feature. The next example shows the application of global variables as a means of communication between two parallel processes.

Example .... Using global variables in a par statement

```
int i, Proc1Busy = TRUE;
par
{{                                              /* process(1) */
   for (i=0; i<Number; i++) {
       DoSomethingWith(i);
       PrintResult(i);
   }
   Proc1Busy = FALSE;
}{                                              /* process(2) */
   while (Proc1Busy) {
       sleep (2);
       printf("Process 1 is in loop nr %d\r",i);
   }
}}
```

In this par statement, a boolean is declared to indicate that process (1) is busy. This boolean is used in process (2), which will run until process (1) has terminated. It uses the other global variable to give a report on the proceedings of process (1) every 2 seconds.



Execution of a for statement

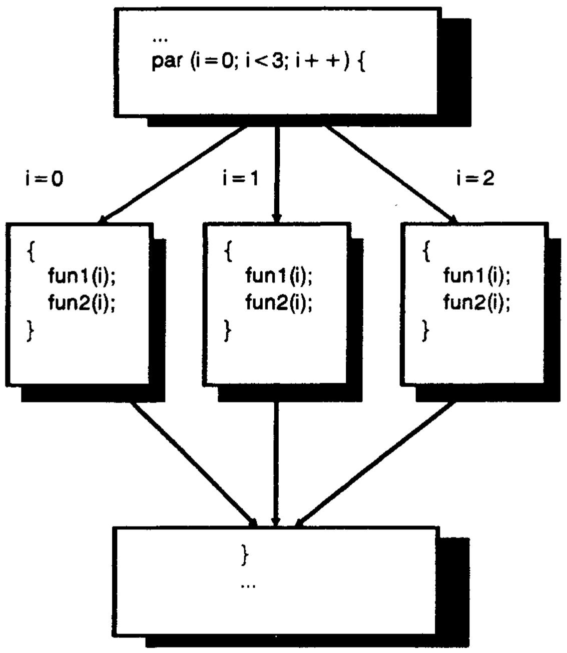
Replicated par statement

The replicated par statement looks like an iterative statement, but operates in a different way. It is important to understand the differences, because errors arising from misuse of a par statement are often very hard to find.

Difference with the for statement

The main difference with an iterative statement is the way the replicator is used. In the replicated par statement, the same code is used by a number of processes which are started to run concurrently, whereas in an iterative statement the same code is executed for a number of times in a sequential order. This difference is illustrated in the two figures.

In the for statement execution of the code part is repeated a number of times, depending on the iteration expressions. There is no change of context, only a change of scope for the single executing process, going through the iteration loops one at a time. Any changes to the iteration variables can be made and will only affect the control flow of this single process.



Execution of a replicated par

In the case of a replicated par statement, the replicator variable is used to start a number of concurrent child processes. These processes are then executed in a time-sharing system, implicating regular shiftings of contexts. The scope rules state that all global variables must be accessible by all child processes, including the replicator variables. But these variables should have different values for each different child process. For this reason, the replicator variables are copied to each of the child process's contexts, initialised to different values according to the expressions in the replicator of the par statement.

Example .... Replicated par statement

```
par (i=0; i<Number; i++) {
   x[i] = i * i;
   y[i] = sqrt(i);
}
```

The par statement listed above will be translated to the following scheme (using pseudo-code to illustrate the behaviour of the par).

Example .... Pseudo-code executed in replicated par statement

```
for (i=0; i<Number; i++) {
   int i', i'';
   i' = i;
   start { x[i'] = i' * i'  } as a concurrent process;
   i'' = i;
   start { y[i''] = sqrt(i'') } as a concurrent process;
}
wait for all processes to end;
```

Replicator versus function calls

To keep track of what happens to replicator variables inside the par statement, a comparison with function calls may be illustrative. In that case, parameters are passed by value, by assignment to the variables which are declared in the funciton definition (and are therefore local to that function). Passing back results can be done in a return value, or by assigning directly to global variables. The values of the locally used parameter variables are lost after returning from the function call.

In the case of a replicated par statement, the "function calls" would be calls to the pieces of code which have been defined as separate tasks inside the par body. But here, no parameters are passed explicitly, and also, the declaration of local variables to hold the "parameter" values is done automatically. Each replicated child process gets local copies of the replicator variables, referred by the same names.

Writing to the replicator variables inside one of the child processes will not lead to an update of the variable by the same name which is known to the parent process. Passing results to the parent process can be done by writing something to global variables. On this point, one should take notice of the remarks made in section "Pitfalls and warnings" below.

Structures and array elements

One last remark must be made on the restrictions of replicator variables that are allowed with the par statement: structure and array elements cannot be used there. The reason is, that allowing them in a replicator of the par statement would cause a conflict of scopes which cannot be solved without interfering with what the average C programmer will expect his code to do.

Execution of a par statement implies multiple processes to be started, which adhere to the same scope rules as elsewhere in C: each of the child processes has access to its own local variables, and to all the variables which are accessible to the parent process. In the case of replicator variables, only the local copies are accessible, since they have received the same names as the global originals.

Allowing structure or array elements as replicator variables would cause a conflict, since part of the structure or array should then be copied to local variables (in order to be able to use their different values), and at the same time the rest of the structure or array must remain accessible at the global level. An example is given below:

Example .... Replicated par using an array element

```
par (a[0] = 0; a[0] < 20; a[0]++) {
   a[a[0]] = ResultOfThis(a[0]);
}
```

The first element in the array is used as an index in the array. The par statement was intended to initialise the other elements. Confusion arises from the use of the same names for the global structure or array and its locally copied element. The solution of copying the complete structure or array to the local level would imply that values cannot be written back to the global structure or array anymore.

It is easy to program around this problem, by using a simple type variable to copy the value of the structure or array element to, before entering the replicated par statement. The example shown above would then look like this:

Example .... Correct version replicated par using an array element

```
int j;
...
par (j = 1; j < 20; j++) {
   a[j] = ResultOfThis(j);
}
a[0] = j;
```

The last line of code is added to obtain the same situation as in the example above, where the upper bounding value of the replicator is also left in `a[0]`.

Pitfalls and warnings

Using globals in nested replicated par statements

The use of nested replicated par statements can lead to problems when using global variables as replicator variables. These problems are directly related to the local copying of the replicator variable, and will be clarified using the following simple example:

Example .... Nested replicated par statements: the wrong way

```
main()
{
   int i,j;
   par (i=0; i<NUMBER; i++) {
       par (j=0; j<NUMBER; j++) {
           printf("%d %d\n",i,j);
       }
   }
}
```

In this example the behaviour of the program is not defined, because it depends on the timing of the concurrent processes, which may vary from one run to the other. The outer par will start NUMBER child processes, but since each of these processes accesses the same global variable `j`, the replication of processes on the inner level will be corrupted. Here the rule of only allowing one process to write to the same shared variable is violated and may cause runtime errors.

This can be overcome in different ways, which will be shown briefly below. The first way is to have variable `j` copied to the local space of the processes started in par statement of level 1, by listing it in the replicator of that par statement as one of the replicator variables.

Example .... Nested replicated par statements: one way

```
main()
{
   int i,j=0;
   par (j,i=0; i<NUMBER; i++) {
       par (; j<NUMBER; j++) {
           printf("%d %d\n",i,j);
       }
   }
}
```

In this example, NUMBER * NUMBER processes are started. Of all variables named in the replicator, local copies are made. In the replicator of the par statement of level 2 copies of j, local to the concurrent processes containing this inner par statement, are used as replicator variables. Since the local copies of i (having the same scope as the local copies of j) are not used in the replicator, these can be used in the way shown.

A more readable and straightforward way of using nested replicated par statements is shown in the following example:

Example .... Nested replicated par statements: another way

```
main()
{
   int i;
   par (i=0; i<NUMBER; i++)            /*   par of level 1 */
       {{                              /* uses copies of i */
           int j;
           par (j=0; j<NUMBER; j++) {  /* par of level 2 */
               printf("%d %d\n",i,j);/* uses copies of j */
           }
       }}
}
```

Each of the processes started in the par statement of level 1 uses a local j, which is copied to local values inside each of the processes started in the par statement of level 2.

Multiple statements inside a replicated par statement

Example .... Replicated par with multiple statements

```
par (p = q; p != NULL; p = p->next)
{
   DoSomethingWith(p);
   PrintResults(p->results);
}
```

It is important to note that all statements defined inside the par body are started as separate processes a number of times. In the example above the possibility exists that a print statement is issued for results that have not yet been calculated. There is no mechanism synchronizing the two functions, which are both executed a number of times concurrently. If only one replicated process is needed, calling first `DoSomethingWith()` and then `PrintResults()`, brackets should be put around both statements to turn them into one compound statement, which is then replicated as a whole. This is shown below.

Example .... Replicated par with one compound statement

```
par (p = q; p != NULL; p = p->next)
{{
   DoSomethingWith(p);
   PrintResults(p->results);
}}
```

The double brackets used in this replicated par statement also provide better readable code when used in a non-replicated par statement. The use of double bracket pairs cause single-statement processes to remain on the same level of indentation as the compound statements listed.

return and continue statements

As is noted in the section on the use of the par construct, the return and continue statements cannot be included in the code inside a parallel process. The reason for these exceptions is that execution of these statements influences the parallel control flow of the program in a way which cannot be handled properly.

In a sequential program, there is only one process executing all of the statements in some order. Using a return statement inside a subroutine causes this same single process to skip the remaining code in that subroutine and resume execution at the level where the subroutine was called. This can also be done inside a for loop without causing problems, since the for statement implements an iteration: returning from one of the iterations implies that the remaining iterations are skipped.

In a parallel program the situation is a little different. Using the return statement inside one of the child processes started in a par construct would imply that this child process should skip the remaining code and resume execution at the level where the subroutine, which contained the par construct, was called.

But this would be an unauthorised change of context for the executing child process: the definition of the par construct implies that child processes may only exist during execution of the par construct. The major effect of having a child process jump out of its context would be the violation of the synchronisation implied in the definition of the par construct.

Using a return statement inside a par construct causes an error during compilation.

The use of the continue statement inside a par construct is prohibited unless this is done inside an iteration which is part of the code of the child process. A continue statement cannot be used to influence the control flow of the parent process (i.e. to skip to the "next" child process), since this would give the child process access to the context of the parent process. Again, since the child process is defined to only exist within the par construct, this access across its own boundaries of existence is not acceptable.

This point can also be seen clearly when realising that the replicated par construct is no iteration in the proper sense: although it is true that the implementation of the replicated par construct involves an iteration to start a number of concurrent processes, this iteration is implicit and not accessible by any part of the user program. The effect is the simultaneous execution of multiple processes, and therefore the continue statement (which is only defined inside the context of an iteration), cannot have a sensible meaning here.

Using a continue statement inside a par construct causes an error during compilation.

The channel datatype

Short description

The implementation of channel as a datatype enables the quick use of communication primitives. Communication between processes is performed in a simple and straightforward way with the channel functioning as a medium between two communicating processes. The data to be transferred over the channel can be of any type. Sending a message to another process is defined as using a channel as left hand side of an assignment operator. Receiving a message is defined as using a channel as right hand value of an assignment operator, or as any rvalue.

Examples of use

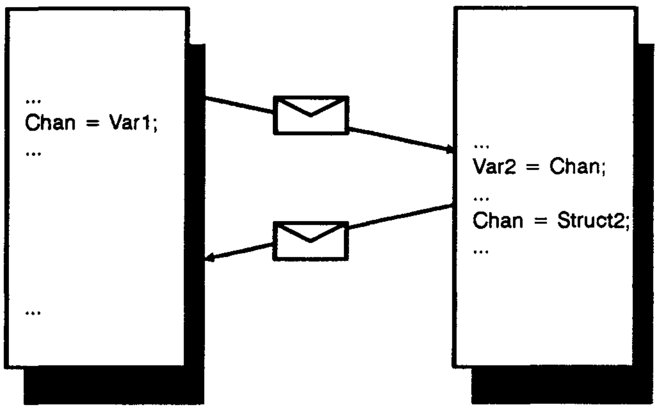
Example ....

```
channel        ChannelName;                              (1)
channel        ChannelArray[SIZE];                       (2)
channel        *ChannelPointer;                          (3)

ChannelName    = Message;                                (4)
Message        = ChannelName;                            (5)
OutputChannel  = (type) InputChannel;                    (6)
```

See also .... The chapter 'Quick reference' for a formal syntax.

Declarations of a channel, an array of channels and a channelpointer are shown in (1) through (3). Communication primitives are defined as assignments, like the statements (4) through (6) shown above, with the channel name on one side and the variable to be sent on the other. The variable to be transmitted and received can be of any type. When redirecting input to output, like in (6), a typecast is needed to indicate the size of the message. Using a channel variable or expression as an lvalue causes a send instruction to be executed; using it as an rvalue causes a receive.

Variable length messages can be transmitted using the functions `_in()` and `_out()`, which are described in the library reference of this manual. Communication via links should be performed by `SendLink()`, `RecvLink()`, `SendLinkOrFail()` and `RecvLinkOrFail()`, also described in the library reference.



Inter-process communication

Use

A channel is used for communication between, and synchronisation of exactly two processes by means of data transfer. A process cannot transmit data to itself. To enable inter-process communication, both processes must use the same channel, and must therefore both know the address of the channel to use. The channel should therefore be global to both communicating processes.

Example .... Communication using a global channel

```
channel Channel;
main()
{
   int  ReceiveValue;
   par {
       Channel = 12345678;
       ReceiveValue = Channel;
   }
}
```

In this example two concurrent processes are started which communicate via the channel declared on level 0 (global to all functions). The same kind of interprocess communication can be set up using channelpointers, as is shown in the next example.

Example .... Communication using channelpointers

```
main() {
   int ReceiveValue;
   channel Channel;
   par {
           Send(&Channel, 12345678);
           ReceiveValue = Receive(&Channel);
   }
}

void Send(To, Value) channel *To; int Value;
{  *To = Value;     }

int Receive(From) channel *From;
{  return *From;    }
```

Scope of channels and passing as parameters

The scope rules for channel variables are the same as for other variables (see the C language definition [Kernighan 1978]). However, channels cannot be passed as parameters in function calls. An attempt to pass a channel as a parameter causes a receive instruction to be executed, and the result of the expected datatransfer to be passed as value to the function called. The number of bytes which will be read is determined by the expected variable type.

Example .... A channel used in a function-call

```
main()
{
   channel C;
   par {
       C = 1000;
       printf("Received %d from channel at address %p\n",
               C, &C);
   }
}
```

Here, the value received (1000) is printed, together with the value of the pointer to the channel.

Detailed description

In all cases with the exception of channel-to-channel assignments, it is clear from the source code what the type of the message to be transmitted and received is. Therefore it is not necessary to indicate the type of the message explicitly. When a typecast is added which is different from the type of the identifier named in the assignment, the type of the cast determines the number of bytes transferred:

Example .... Typecast and conversion in communication

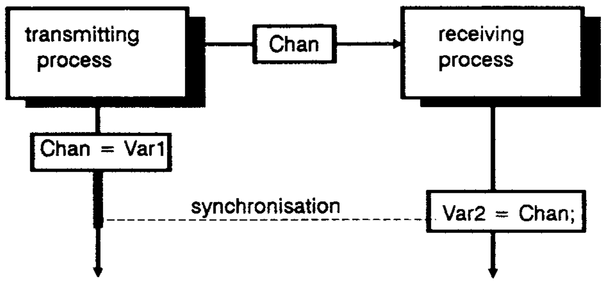
```
char i;
channel c;
i = c;           (1)
i = (int) c;     (2)
```

In statement (1), one byte is received and assigned to `i`. In statement (2), four bytes are received and converted to one byte before assigning the resulting value to `i`. In all assignments involving channels the rule is, like in all C assignments, that the righthand side is evaluated and the result is then assigned to the variable identified on the lefthand side.

Communication and synchronisation

Channel communication is always point-to-point, which implies that the communication can only take place when both processes are ready for it. The process that reaches the transfer-point first is descheduled until the other process is ready for the transfer as well. At that moment, the communication will be executed and both processes are able to proceed. In this way, communication between two processes serves as a point of synchronisation. The diagram below illustrates this aspect of communication.In this diagram the process on the left has to wait for the other process to arrive at the synchronisation point.

To obtain an asynchronous communication between two processes, a buffer process should be added, which takes over the waiting time and allows the first process to arrive at a communication to proceed immediately, without having to wait for the destination process to arrive at the corresponding communication point. The buffer process uses a temporary variable to store the data received from the first process, and can be coded very easily using a channel-to-channel assignment.

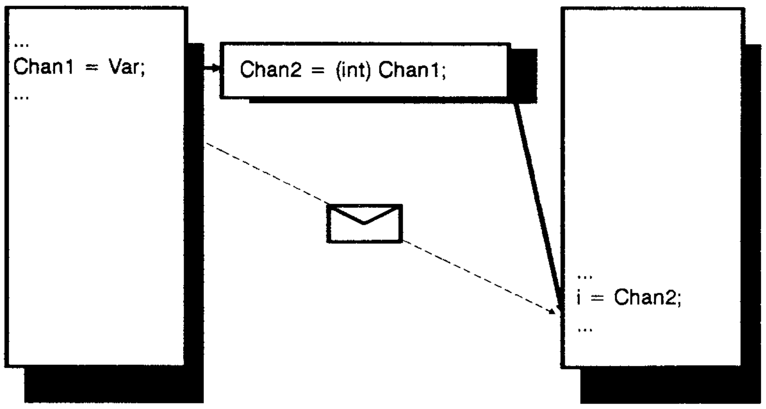


Synchonisation of channels at communication

Example .... Buffer process in parallel C

```
for (;;)
   ChannelOut = ChannelIn;
```

However, from this assignment the compiler cannot know which type to store and forward. Therefore, the programmer must add a typecast to indicate the type to be expected, as in the earlier example. When the type is not specified, an integer is assumed.

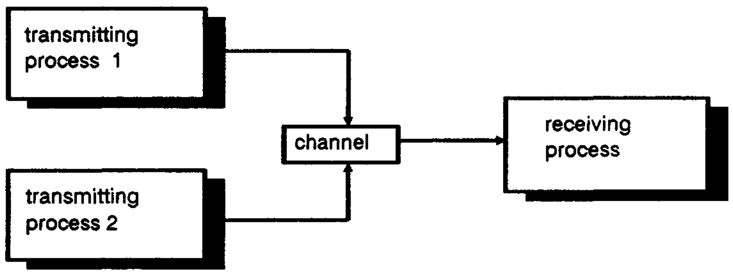


Buffer process using typecast

Using linkconnections between transputers

The links of the transputer behave like channels, and can be used by initialising a channelpointer to the address of a link. However, it is preferable to use the functions `SendLink()` or `RecvLink()` described in the library reference of this manual. This will provide upward compatibility with future versions of the Par.C System (possibly including automatic multiplexing with the message passing system which is currently under development).

Pitfalls and warnings



Multiplexing on a channel

"Multiplexing" communications on one channel

Channels are defined as point-to-point connections between processes. An attempt to use the same channel for simultaneous communication between more than two processes will lead to processes hanging up and should therefore be avoided. The situation depicted in the above figure can easily be implemented by shielding a channel with a semaphore (See the runtimefunctions `P()` and `V()`). An alternative implementation of the above would be using two different channels, one for each of the sending processes, and a select statement in the receiving process.

Using "malloced" channels

When channels are declared in memory which is allocated using `malloc()` or `smalloc()` these channels must be reset before using them for the first time. This is done by a call to `ResetChannel()` with the channel-address as parameter. To reset an array of channels, the function `memfill()` can be used (see library reference). The value to store on the channel-address in order to reset the channel is `0x80000000`, defined as `MostNeg` in `stddef.h`. Another possibility to reset a channel is using the following expression:

```
*(int *) &Channel = MostNeg;
```

Non-equal size and direction of communication

Data transfer over channels will only be successful when the corresponding channel assignments are complementary (the same direction of data transfer) and the size of the message corresponds. If differences in either of these occur, the results are as follows (depending on whether internal channels or external links are used):

- With internal channels, the second process to arrive at the communication point determines the size and the direction of the data transfer. An error arising from non-equal message directions or sizes in corresponding transmission and reception statements will not be detected by the communicating processes, nor by the transputer hardware. Both processes will simply resume execution after the erroneous data transfer has taken place, although at least one of them has incorrect data to work with. Therefore, these errors may be very hard to figure out.
- Making errors in size or direction when communicating over links, will cause one or possibly both of the processes to crash. With a difference in message size, one of the processes will finish its communication instruction and resume execution, while the other process keeps waiting for the last bytes to be sent from, or accepted at, the other side. When the link is accessed for output or input on both sides, the processes will never gain contact (because of the division of each link in an outward and an inward bound channel).

The programmer should therefore take great care in checking on the symmetry of corresponding communication statements. One of the methods to help in having corresponding message sizes is using a typedefined message structure, which can then be used to define local structures to send and receive over channels.

The select statement

Short description

The select statement is somewhat similar to the switch statement. A switch associates pieces of code with specific values of an expression, whereas the select associates pieces of code with specific events. A 'guard' is set up for each of the events defined inside the body of the select statement. The first event to occur triggers execution of the associated code.

Examples of use

Example .... Simple select

```
select {
   alt guard <channelpointer>:     <code>
   alt guard <channelpointer>:     <code>
   ........
   alt guard <channelpointer>:     <code>
   alt cond <expression>:          <code>
}
```

Select with timeout

```
select within <expression> {
   alt guard <channelpointer>:     <code>
   alt guard <channelpointer>:     <code>
   ........
   alt guard <channelpointer>:     <code>
   alt timeout:                    <code>
}
```

Other alternative clauses

```
alt <replicator> guard <channelpointer>
alt cond <expression> guard <channelpointer>
alt cond <expression> timeout
alt <replicator> cond <expression> guard <channelpointer>
```

See also .... The chapter 'Quick reference' for a formal syntax.

A select statement consists of the keyword `select`, followed by the select body enclosed in braces. The select body consists of a number of 'alternative clauses', indicated by the keyword `alt`. Each alternative indicates an event which may occur and the code that should be executed on that occasion. The first of the specified events to occur will 'trigger' the selecting process, which will then execute the code specified for this particular event. The select body should contain at least one alternative clause.

The select is able to detect the following events:

- another process is ready to output over a channel,
- a specified time limit is exceeded,
- the event pin of the transputer is asserted.

A 'guard' is raised for each event. The keyword `guard` must be followed by a pointer to a channel.

Timeout

A maximum waiting time for the selecting process can be indicated using the `within` keyword in the header of the select, followed by an integer expression. The value of this expression is the number of clockticks within which one of the events must occur. Otherwise, the `timeout` event will be triggered.

Replicator

Alternatives may include a replicator, causing a number of events to be waited for, without having to write out the same alternative clause for each of them. The replicator variables can also be used in the code specified in the replicated alternative. The replicator is not allowed in a timeout alternative.

Conditional part

Also, alternatives may include a conditional part, consisting of the keyword `cond`, followed by a boolean expression. This expression is evaluated and the result determines whether or not the specified event will be included in the select. The conditional part can be used in combination with a replicator, and may then include the replicator variable(s) in the boolean expression.

Note . . . . The condition may be `FALSE` when setting up the guards in the select, but become `TRUE` while the selecting process is waiting. This will not cause an extra guard to be included, and may even cause erroneous behaviour after one of the other events has triggered the select. See the section on "Side-effects" in the paragraph "Pitfalls and Warnings" of this chapter.

Code

The code specified in an alternative clause is placed behind the colon, and may consist of any series of statements. The code may also consist of an empty statement list, but it should be noted that omitting the break instruction at the end of a piece of code causes the selecting process to 'fall through' (like in a switch statement), into the code belonging to the next alternative.

Use

In a parallel program, with a number of concurrent processes running simultaneously, the order in which processes will reach certain points of synchronisation with other processes cannot be determined before the program is actually run. And even then, from one run to the other, changed values of variables may influence the timing of processes dramatically.

In this situation, the select statement provides an instrument to influence the parallel control flow of the program in a more dynamic way than would be possible using only process-to-process channels, or signal handling routines. Since the selecting process waits for a number of events at the same time, the varying order in which events occur can be handled by the program.

Multiplexing channels

The most obvious situation in which a select statement can be used is a multiplexer of a number of communication channels. This situation is depicted in the following figure. Using channel communications, a simple 'client-server' model can be implemented. The differences between using and not using the select statement for this are explained in the examples below.

Consider the following situation: process `A` and process `B` both need the service of process `S`. Requests are made by sending some information over channels to `S`. `A` and `B` use the channels `chan_A` and `chan_B` respectively. A naive implementation of s would look like this:

Example .... Service without select

```
while (!Ready) {
   ProcessInfoFrom (&chan_A);
   ProcessInfoFrom (&chan_B);
}
```

This code is purely sequential. Process A will be served first, and then Process B. If B happens to be the first with a request for S, it will have to wait until a request by A has been received and processed. Moreover, if A does not do any requests, B will never be served.

A more robust implementation, using the select statement, is listed below.

Example .... Service with select

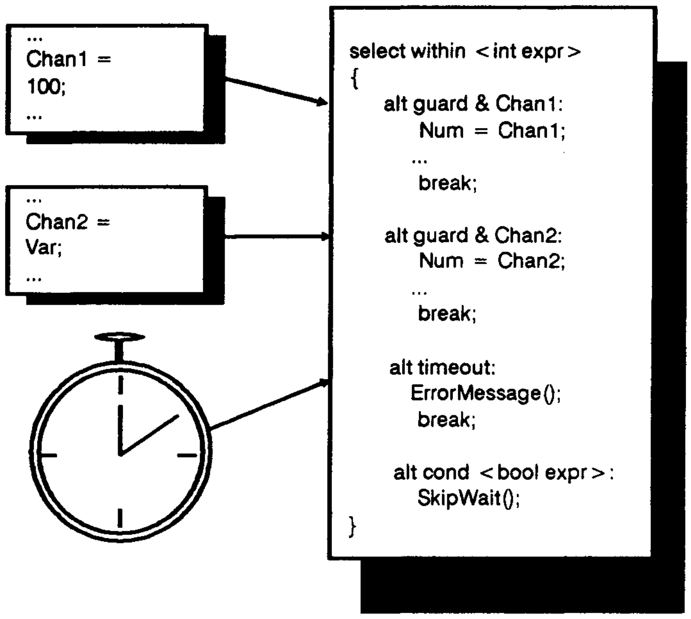
```
while (!Ready) {
   select {
       alt guard &chan_A: ProcessInfoFrom (&chan_A); break;
       alt guard &chan_B: ProcessInfoFrom (&chan_B); break;
   }
}
```

The select statement selects code indicated in the `alt` (alternative) clause which matches the triggering event. In the above example, the event is a process becoming ready to send data over an indicated channel. The keyword `guard` precedes a pointer to the channel which has to be guarded. Process S now looks at both channels at the same time and serves the first process to send a request.

Specifying a timeout value

The select includes the possibility of specifying a maximum waiting time, e.g. in order to prevent deadlock situations. This can be done using the `within` keyword in the header of the select, followed by an integer expression. The resulting value of this expression is taken as the maximum waiting time, measured in clockticks from the moment the select is entered.

In this timed select statement, a 'timeout alternative' is allowed, specifying code which should be executed when the 'timeout event' occurs. The use of this alternative is shown in the example below.



Combining multiple input channels

Example .... Timed select

```
select within MaxWaitTime {
   alt guard &InputChannel: Var = InputChannel; break;
   alt timeout: InputFailed = TRUE;
}
```

The timeout alternative cannot be used without the `within <int expression>` part in the header of the select statement. Also, it is illegal to use more than one timeout alternative, or a replicator in one single timeout alternative.

It is possible to switch the timeout alternative on and off during runtime using a conditional expression, as is shown below:

```
alt cond TimeoutEnabled timeout: break;
```

This will cause a timeout event only if `TimeoutEnabled` is `TRUE`.

Replicated alternatives

Each of the alternatives in the select statement, with the exception of the timeout alternative, may include a replicator of the form shown in line (3) in the syntax section of this chapter. This will cause a number of the same alternatives to be set up. The channelpointers used are calculated using the replicator variable(s).

Example .... Replicated alternatives

```
channel c[4];
channel multiplexed;
int dummy;
select {
   alt (i=0; i<4; i++) guard &c[i]:
        dummy=c[i];
        multiplexed=dummy;
        break;
}
```

This code shows a simple multiplexer process. It guards the four channels in the array `c[]`, using the replicator following the `alt` keyword. The first of these channels which has data available will trigger the select. The selected code will be executed with the value of the replicator variable `i` indicating the triggering channel `c[i]`. The process reads an integer from this channel, and forwards it over the channel `multiplexed`. In this way, four outputs are connected to one input. Note that the use of the replicator keeps the code concise and readable, since we do not have to duplicate the alternative clause for every channel to guard.

For details on the side-effects in replicators that should be avoided, see "Pitfalls and warnings".

Conditional alternatives

It is often desirable to avoid an event to be waited for, i.e. if a channel is not always used to communicate with the selecting process. In some cases, the select statement may be executed a number of times with different lists of events for each time. This can be accomplished by adding a conditional part to the alternative clause(s).

In the following example this feature is used to guard a number of channels contained in structures which are kept in a linked list. The end of the list is indicated with a `NULL` pointer. The nodes in this list have a field indicating the 'owner' of the channel. Only those channels which are owned by `SERVER` should be guarded.

Example .... Use of the conditional specifier

```
typedef enum {SERVER, IO_DRIVER, MESSAGE_PASSER} RESOURCE;

typedef struct grd {        /* define the list            */
   channel *chan;               /* pointer to channel     */
   RESOURCE owner;              /* owner of the channel   */
   struct grd *next;            /* linked list pointer    */
} *LIST;

extern LIST SysChans;           /* list of system channels*/
LIST s;                         /* scanning pointer       */
int ReqCode;                    /* Action to be done      */

select {
   alt (s=SysChans; s!=NULL; s=s->next)  /* scan the list*/
   cond s->owner==SERVER        /* if it's mine:          */
   guard s->chan:               /* guard it               */
                                /* when selected:         */
       ReqCode = *s->chan;      /* determine what to do   */
       ServeRequest(ReqCode, s->chan);  /* do it          */
       break;
}
```

Note the use of the replicator. It scans a pointer through the linked list of system channels. The selected code will be entered with the variable s pointing to the right node of the list. The condition that only channels owned by SERVER should be guarded is expressed in the condition following the replicator:

```
cond s->owner==SERVER
```

The condition only serves to alter the set up of the select. Once the process is set waiting, only events can trigger it. Change of a condition is NOT an event, so the select can not be used to suspend a process until a certain condition becomes TRUE. Moreover, in the current implementation of the Par.C System, changing conditions while a selecting process is waiting will cause undefined behaviour, and should therefore be avoided. Details on this are given in the paragraph "Pitfalls and warnings".

'Default' alternative

The condition in an alternative can also be used without the keyword guard. In this way the select can be kept from waiting for an event, in case none of them was active before the select was entered. This can be seen as the counterpart of the 'default' label in the switch statement. Note however, that this 'default' alternative can be switched on and off during runtime.

Example .... Using a default alternative

```
int HasInput(C)
channel *C;
{
   select {
       alt guard C:      /* Triggered if input available   */
           return TRUE;
       alt cond TRUE:    /* Always triggered, except when */
           return FALSE; /*  event above occurred          */
   }
}
```

In the above example, the selecting process will only return TRUE when input is available on the specified channel before the select is entered. Otherwise the default alternative is triggered.

Another use of the 'default' alternative is shown in the detailed description section, in the paragraph on priorities.

Detailed description

There are two levels of code in a select statement. The first level, the selecting code, is used to set up the select, and again to determine which of the events has occurred just after the select has been triggered. This level includes all code in replicator, conditional and guard expressions. In the analogy with the switch statement, this code represents the expression in the header of the switch, together with the case and default label indications.

The second level, the selected code, is only executed after the selection has been made. At that time, all first-level code has been executed, and the only result of this is a value for the replicator variables. This selected code is everything which follows the colon up to the next alternative or the closing bracket of the select statement.

It is important to separate these two levels when writing code, to keep a clear view on what is supposed to happen. This separation is not entirely obvious, because a larger part of the code is placed in the different alternative clauses than in the switch (where the cases represent label values).

Explicit communication

The guards used in the alternatives trigger the select without actually performing an input. The statements defined in the selected alternative determine what is done with the active channel. Receiving a message from the active channel should be done explicitly.

It follows that one can also defer the handling of input to another process. If we use the function `HasInput()` from the previous example we could for example write:

Example .... Explicit channel communication after select

```
channel inputChan;

if (HasInput(&inputChan))
        ProcessInput(&inputChan);
```

The function `ProcessInput()` might be used to read input from the channel or to signal another process to do it.

Links and events

In principle, links and channels are of the same kind. The only difference on the transputer is in their placement and the fact that only a limited number of links are available per transputer. Channels are not placed explicitly: they will be allocated somewhere in memory whenever they are declared. Links are situated at the bottom of the transputer's internal memory.

In the Par.C System, `LINKIN()` and `LINKOUT()` macro's are provided to place channels on link addresses. The `LINKIN()` macro can be used directly in the select statement, to guard on one or more input links. The macro `LinksPerTransputer` indicates the number of links that can be used. In the same way, the `EVENT` macro can be used to indicate the specific memory address, which behaves like a channel and corresponds to the transputer's event pin. Note that this 'event channel' can only be used in the select statement to look if the event pin is asserted.

Example .... Using the event pin

```
#include <transp.h>
select {
   alt guard LINKIN(0):          /* check for input on link 0 */
       ReadFromLink(0);
       break;
   alt guard EVENT:                    /* check the event pin */
       HandleInterrupt();
       break;
}
```

Note that these macro's operate on the lowest level of the transputer hardware. They might interfere with other runtime functions. The file I/O system will use all links in the direction of the host. See also `sendLink()` etc. Reading from a link occupied by the file I/O system almost certainly corrupts the runtime system. EVENT may also be handled by one of the 'signal' facilities. See `signal()` for a detailed description.

Priorities

The definition of the select implies that one event (the first to occur) triggers the selecting process into executing the code specified for that event. In some cases, multiple events may become active at the same time. This brings up the point of deciding which code should be executed in such situations.

To indicate the importance of this point it should be noted that, on the transputer, triggering the selecting process does not necessarily imply its immediate reaction: the process will be descheduled, and eventually become active to decide which actions must be taken next. In the meantime, more events may have occurred, and there is no way the selecting process can know which one was first.

When the selecting process resumes execution, the list of alternatives is checked in the order in which they are listed in the source code. Thus, if more than one event occurred, the code specified for the first one listed will be executed.

This opens up possibilities for a special kind of use of the 'default' alternative: if one wants to include or exclude a whole list of events at once, it is not necessary to include the same condition in all of the alternative clauses specifying these events. One can simply put a 'default' alternative before the group of alternative clauses. If the condition evaluates to TRUE, this will cause this whole group to be skipped, since the 'default' alternative has a higher priority than all events listed below it.

Example .... Using the 'default' alternative to skip a group of alternatives

```
select {
   alt guard Channel1: code1(); break;        /*first group*/
   alt guard Channel2: code2(); break;
   .....
   alt cond Jump_Out1: break;
   alt guard ChannelA: codeA(); break;        /*second group*/
   alt guard ChannelB: codeB(); break;
   .....
   alt cond Jump_Out2: break;
}
```

This select cannot be triggered by any of the alternatives in the second group if `Jump_Out1` is TRUE. The effect of this is to skip all events following this 'default' alternative. As is shown, it is possible to include multiple 'default' alternatives in the same select statement, even if all of them are enabled by their conditions being TRUE. Note that in this example one of the channels in the first group only triggers the select when it is active before the select is entered, otherwise the `Jump_Out1` alternative is selected immediately.

Example: round-robin selecting mechanism

To show the possibilities of the select statement, this section contains an example of a 'round-robin' selecting mechanism. The situation can be pictured as follows:

A message passing process is connected through a number of channels to processes which try to send data. The selecting process will immediately re-enter the select, after a message has been passed on, using the same list of channels.

On the average, each process will send the same number of messages with the same random delay between two messages. If we would simply list all of the channels in one specific order, the channels specified in the first alternative clauses would be served more times than the ones listed at the end.

To avoid this situation, the channels are placed in structures, which are connected into a circular list. Two replicator variables are used: an integer to keep track of the number of channels which are included in the list of input events, and a pointer to scan the circular list of structures. After a channel is served, this pointer is initialised to indicate the next structure. In this way, priorities are equalised over all channels in the list.

Example .... Code for a round-robin select.
The types from one of the previous examples are used.

```
int i,ReqCode;
LIST p,q;

while (!Ready) {                              /* loop until ready*/
   select {              /* guard the appropriate channels */
       alt (i=0,p=q; i++<COUNT; p=p->next)
       cond p->owner==SERVER guard p->chan :
           ReqCode = *p->chan;
           ServeRequest(ReqCode,p->chan);
           q = p->next;            /* start following select */
                                   /*       at the next node */
           break;
   }
}
```

Nested select statements

Since the code following an alternative may contain any statement, including a select, the nesting of select statements is possible. The example listed below serves to illustrate this:

Example .... Nested select statements

```
select within 1000 {
   alt guard ChanPtr1:
       select within 1000 {
           alt guard ChanPtr2:
                printf("Both channels active  \n");
                break;
           alt timeout:
                printf("Only channel 1 active \n");
       } break;
```

```
    alt timeout:
        select {
            alt guard ChanPtr2:
                printf("Only channel 2 active\n");
                break;
            alt cond TRUE:
                printf("Both channels inactive\n");
        }
}
```

It is important to see that the nested select in this example is only reached after the select on level 1 has been triggered by a process having output for channel 1. This implies that channel 2 will only be guarded after channel 1 has become active within 1000 clockticks after the select at level 1 is entered. A timeout occurring in select on the outer level does not imply anything about the availability of input on channel 2.

Pitfalls and warnings

Using channels awaiting output

The instruction set of the transputer only allows guarding input channels. When a guarded channel becomes active because a process tries to read from that channel, the effects are unpredictable. Therefore, the programmer should take care that channels which are being guarded in a select statement are only used by other processes to write to.

Of course, in the second `HasInput()` example, when the function `ProcessInput()` is used to signal another process that it can read from the channel, care should be taken to ensure that the other process is ready with the input before the channel is used again in a select.

Side-effects

Execution of a select statement starts with a loop in which the guards for all specified events are set up. After this first loop the process executing the select statement is temporarily halted, until one of the guards triggers the select. The process then again executes the same loop which was performed in setting up the different guards, to check which one has triggered the select.

When using side-effects in the expressions in the conditional part of the alternative clauses defined within the select, the second loop executed may not be entirely the same as the first one. This may cause errors to occur, since a guard which has triggered the select may be evaluated as being non-valid in the second run, and remain unnoticed in this way.

The programmer should refrain from using any side-effects in the first level code. Only when both passes of this code will yield the same results, will the selecting process behave properly.

Using replicator variables

Another important fact to note is that only the replicator variables are guaranteed to have the correct value when the selected code is entered. If side effects take place on objects pointed to by replicator variables, things might not come out as expected.

Example .... Incorrect use of replicator variables

```
int *p;
int x;
channel c[10];

select {
   alt (*p=0; *p<10; (*p)++) guard &c[*p] :
        x=c[*p];
        break;
}
```

In this example, p is the replicator variable. The compiler ensures that the value of this variable is correct when the selected code is entered. However, not the value of p, but the value pointed to by p is important in selecting from which channel to read. This value is not preserved, and so x will be read from the wrong channel. (To be precise, *p will have a value of 10 at the end of the loop, and therefore x will be read from c[10], which is not a valid channel at all).

Tutorial

Introduction

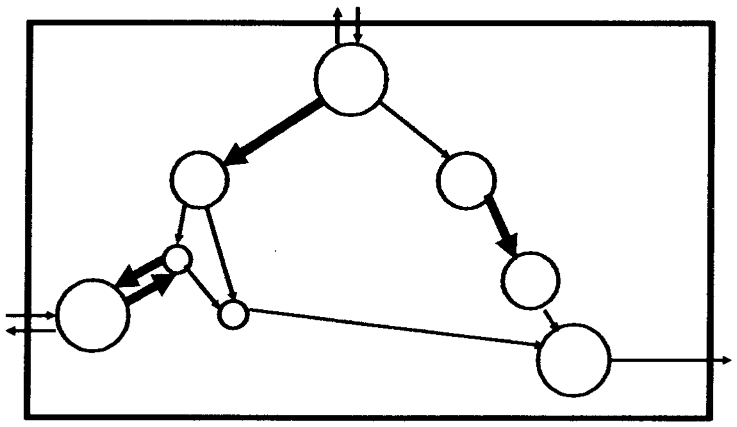
This part of the manual explains how to use the Par.C System to write parallel programs. First, an introduction into parallel processing is given. The concept of dynamic parallel processing, and the support for this provided by the Par.C System, is addressed then, followed by an example of a dynamic parallel program.

Although this tutorial is not nearly as extended as we would like it to be, we believe it will prove useful to users of the Par.C System to get acquainted with some of the basic concepts used in the design of the system.

Introduction to parallel processing

Parallel processing involves the division of work between a number of concurrent processors. To many people, this may seem to be a burden: programmers for the most part have been trained to write software in a sequential way, since the target systems mostly contain only one single CPU. Still, the human organism is a highly parallel "machine", and working with parallel programming for some time shows that we are indeed capable of dealing with parallel problems quite easily.

Parallelising the execution of a certain task implies that the inherent possibilities for parallelism in the task have to be found and mapped onto physically distinct processes or processors in an efficient way. Therefore, the first step in writing a parallel program is to draw a logical map of the problem, dividing it into a number of tasks which communicate over interconnecting channels. For a typical problem, the logical map may look like shown in the figure below.



Logical map of a parallel program

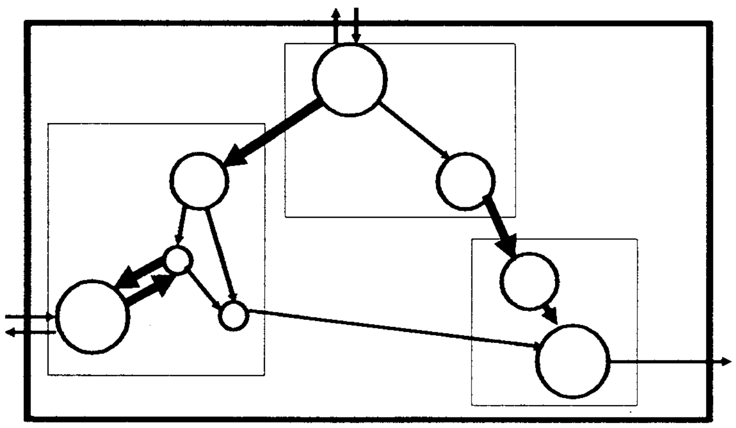
Mapping this logical structure onto a number of concurrent interconnected processors implies that certain conditions must be met, according to the possibilities the target hardware provides. In the case of transputer systems, the hardware consists of general purpose processors with local memory to store code and data, and interconnecting links for datatransfer.

The fact that not only processing power but also memory is distributed over the system, implies that tasks residing on different processors will not have access to the same data areas directly. This implies that when multiple tasks need access to the same data, either the tasks have to be placed onto the same processor, or data has to be communicated between tasks on different processors.

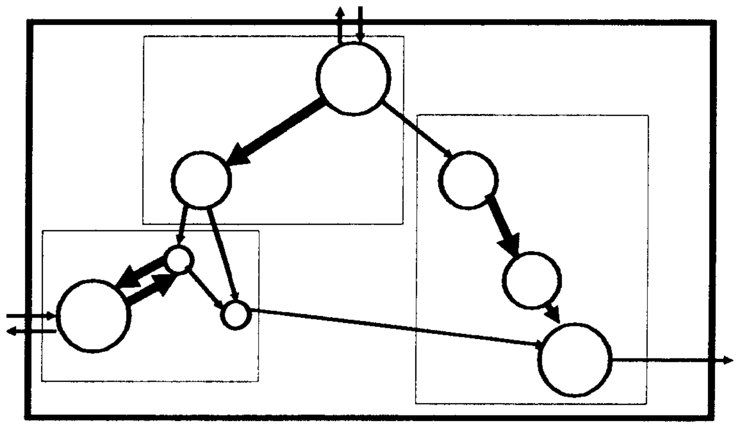
But there is more involved when mapping the logical map of a program onto a parallel system. The main purpose of parallelising a program is of course to obtain a higher performance. When using transputer systems, the speedup can be almost linear with the number of processors. However, not only the increase in total processing power of the system counts: with more processors working on the same parallelised job, communication between processes running on different transputers will inevitably increase.

Although communication takes place in parallel with calculations being executed by the CPU, it will always take some time before the data arrives at the destination, especially when processes on remote transputers are interconnected through a number of intermediate processors. Therefore, the ratio of calculation time over communication time determines the performance of a transputer system.

This opens two strategies to map the logical structure of the program onto the physical structure of the available transputer system. The first divides the tasks over the system in such a way that each of the processors will have approximately the same workload, depicted by the different sizes of (process) circles. Using the logical map of the previous figure, this can be done as follows:


Equally divided workload per processor

In this case, the differences in time needed for data communication on the same processor and between remote processors, as visualised by the thickness of the arrows connecting the processes, has not been taken into account. Especially in cases where inter-process communication is dense, an increased performance may be obtained by placing these "tightly coupled" processes on the same processor. In most applications, a trade-off between equally divided workload and minimised inter-processor communication will prove to offer the best results.



Mininised inter-processor communication

Of course, hardware constraints may influence the possibilities for efficient mapping in more or less severe ways. For instance, a processor on a graphics board will have to handle the graphics commands. This may imply that no other task will fit into its memory, and minimising inter-processor communication means that the tasks to support the user interface have to be placed on the transputers neighbouring this graphics processor.

Dynamic parallel processing

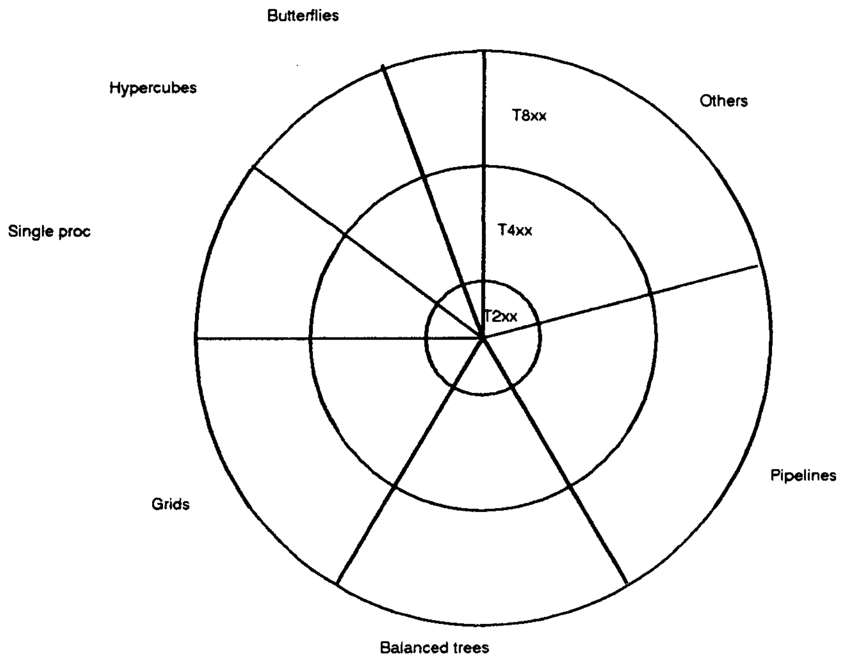
One of the most eye-catching characteristics of transputer systems is their flexibility: virtually any number of transputers can be connected into systems of varying configurations. This enables the production of computer systems which are exactly tailored to suit the needs of certain applications in terms of processing power. Also, expanding processing power simply means adding more transputers to the system. A nearly linear speedup can be obtained, provided the program is designed with this option in mind.

With this flexibility as a main characteristic of transputer systems, another view on parallel programming emerges, which distinguishes between static and dynamic approaches. These are defined in terms of the target systems for which the parallel program is designed: if a program is destined to run on one specific transputer system, it can be characterised as a completely static parallel program; if the same program should run on a wide variety of target systems (varying in topology and size), the program must be parallelised in a dynamic way.

In fact, there is no clear distinction between two classes of parallel programs using the static/dynamic characterisation: programs can be placed somewhere on a scale between the two extremes. The position on this scale is determined by the presuppositions which are made about the target systems for the program. Each presupposition narrows down the number of actual systems on which the program can be run, i.e. makes the program less dynamic. On the other hand, presuppositions enable the programmer to simplify the program structure and to increase performance, for instance by using a faster and simpler scheme for message passing in a grid configuration or a hypercube.

To indicate how presuppositions on the target system hardware narrow down the range of systems on which the program can be executed, an overview can be drawn showing the division of transputer systems over transputer types and architectures.

The rings are used to divide transputer systems according to the constituing processor types. Joining two or more rings produces a class which contains "pure" systems of each of the types plus any mixed system which can be built by including transputers of all indicated types. The influence of the types of transputers contained in the target systems for a program can be shown quite clearly: when producing a program destined to run on a network of T8xx's only, the use of special floating point code and math libraries increases performance dramatically. In the case of a mixed T4xx/T8xx network, one can choose between full transparency of the code or division of tasks between the different types according to whether floating point arithmetic is used or not.



Overview of different transputer system configurations

The above figure also shows a subdivision of all transputer systems into commonly used architectures: butterflies, hypercubes, single processor systems, grids, balanced trees, pipelines, and others.

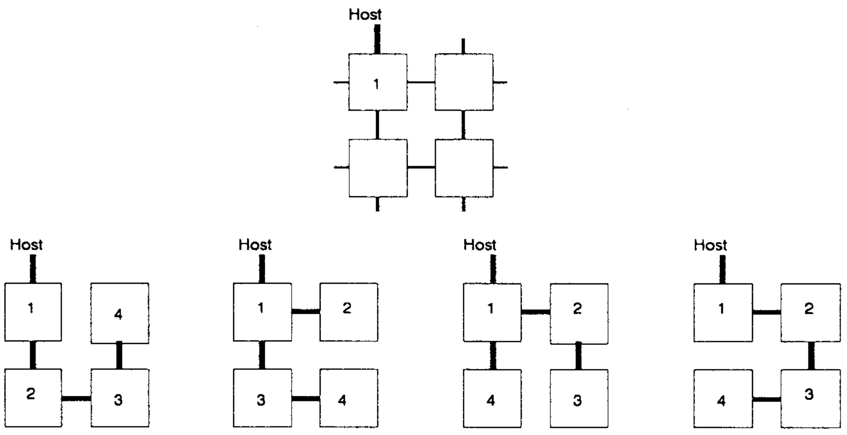
The influence of different architectures can be exemplified in the inevitable message-passer facility inside the program: some architectures allow much simpler and therefore possibly faster message-passing schemes than others. In the case of a single-processor system, messages will be passed directly between processes on the same processor, using a memory word as a "soft channel". In the case of non-balanced trees without connecting links between different branches, messages to a processor residing in another branch will have to travel up to the bifurcation from where they can descend towards their destination. Of course, the aptness of a certain architecture for a certain parallel program is highly determined by the structure of the problem itself.

The Par.C bootsystem

The Par.C bootsystem is able to load the same program on transputer networks of different sizes and configurations. This is done on the basis of a network analyser, which searches the available network every time a program is loaded (note that this happens only when linking `b.lib`).

This analyser acts as a loader for the network, and leaves its information behind for the program to use. Since an important part of this information depends on the search strategy and the resulting boot-tree, this strategy is briefly explained here.

The search executed by the network analyser is completely parallelised, resulting in a boot-tree which grows broader as fast as the interconnections between the transputers allow. In this way, the shortest path from the first transputer to each of the other available processors is found. Note that this path is the shortest in time rather than in number of links or 'hops' needed to reach each of the nodes: it may take more time to pass three transputers running on 15 MHz with 10 MBit/s links than to pass five 25 MHz nodes communicating at 20 MBit/s. In some cases, the timing of two or more different bootpaths differ to such little extent, that different boot-trees may emerge from rebooting the same network.



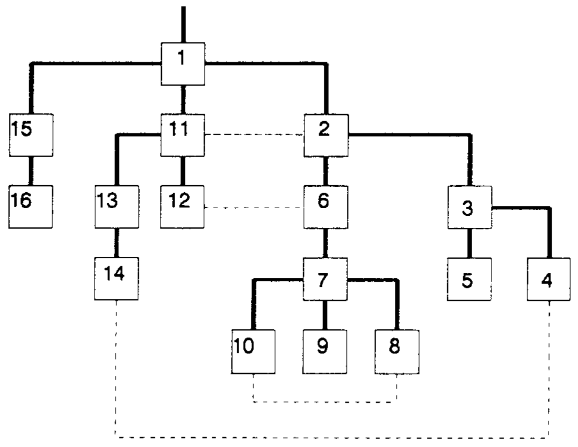
Possible boot-trees in a four-node network

When a program needs to perform different tasks on different processors, without losing the adaptability of the software to the available hardware during loadtime, the identity numbers given to each of the nodes can be used. The scheme by which these numbers are given is explained below.

The numbering scheme used by the bootsystem uses the information which is already available in the boot-tree representation in each of the nodes. The first transputer, after receiving the number of transputers connected to each of its links, sets its own identity to 1, and hands out the number of its 'child nodes' as shown in the pseudo-code below:

Pseudo-code.....
```
read ParentIdentity from the link uptree
set OwnIdentity to (ParentIdentity + 1)
set ChildIdentity to OwnIdentity
for each link
    if this link is connected and going downtree
        send ChildIdentity down this link
        add the number of transputers connected via this
         link to ChildIdentity
```

The SYSTEM structure



Numbering of transputers

When loading a Par.C-produced program into a transputer network, the complete code of the program is copied to, and started on, all processors. Since the code is not spread over different nodes no predefined configuration is needed. Instead of this, the execution of the code is influenced during runtime, based on knowledge of the actual system the program is running on. This information is left behind by the booter on each node, and becomes available to the user program in a SYSTEM structure after calling the function `GetSysInfo()`. The contents of this SYSTEM structure are given in its declaration in `system.h` as shown below:

Definition .... The SYSTEM structure

```
typedef struct _system {
   _Word   HostLinkno;       /* Number [0-3] of the bootlink/
   channel *HostLinkOut;     /* send uptree                */
   channel *HostLinkIn;      /* Input from uptree          */
   _Byte   *MemStart;        /* lowest available address   */
   _Byte   *MemTop;          /* highest available address+1*/
   _Word   ProcessorSpeed;   /* in 250kHz units            */
   _Word   XMemSpeed;        /* expressed in proc.cycles   */
   _Word   ttype;            /* 2, 4 or 8                  */
   _Word   Tn;             /* Identity nr within network   */
   _Word   nT;             /* Number of T's in the network */
   _Word   nT_Down;        /* number of T's in our branch  */
   _Word   NBooted[4];       /* Active T's on each link    */
   _Word   Network[4];       /* Neighbours Tn              */
   _Byte   LinkStatus[4];    /* linkstatus                 */
   _Byte   ExtLink[4];       /* Links of neighbours        */
   _Byte   ExtType[4];       /* Types of neighbours        */
} SYSTEM;
```

An accompanying pointer type is also defined:

```
typedef SYSTEM *System;
```

The meaning of the fields is as follows:

HostLinkno

Number of the link by which this transputer was booted. Normally indicates the fastest path to the host.

HostLinkOut

Channelpointer connected to the HostLink. Through this channel data can be output to the HostLink.

HostLinkIn

Channelpointer connected to the HostLink. Through this channel data can be input from the HostLink.

MemStart, MemTop

Boundaries of the available contiguous memory. Note that addresses of the Inmos transputers are signed, so these pointers probably have negative values.

ProcessorSpeed

Gives the processor clock speed in 250kHz units.

XMemSpeed

Speed of the external memory expressed in the number of processor cycles required to access a memory location.

ttype

Indicates the generic type of transputer. For various transputer types values have been defined in system.h. The T414 for example has type 4, and the T800 has type 8. The type code is intended to reflect characteristics of the transputer, such as the word length and the presence of a floating point unit.

Tn

The unique identity number of this transputer within the network. Each transputer is given a unique number to identify it. The algorithm by which the numbers are assigned was explained earlier. The transputer connected to the host has identity number 1.

nT

The total number of transputers booted in this network.

nT_Down

The number of transputers in this branch of the boot tree, including itself.

NBooted[]

The number of transputers booted through each link. Note that the sum of the elements of this array is one less than `nT_Down`. Also note that `NBooted[i]` is zero if:

- `i` is the number of the HostLink.
- The transputer connected to link nr `i` was already booted through another path.
- There is no transputer connected to link nr `i`.

Network[]

The identity numbers (`Tn`) of the transputers connected to each link. For links with no transputer connected it has the value `NOT_CONNECTED`. The transputer which is connected directly to the host has the value `HOST_CONNECTED` in the element corresponding to the HostLink.

LinkStatus[]

Gives specific status information for each link. Values can be found in `system.h`.

ExtLink[]

Holds, for each of this transputer's links, the number of the link as seen from the neighbour to which it is connected.

ExtType[]

The ttypes of the neighbours connected to each link.

The boot-tree is represented on each node by information of its links. The link from which the node has been booted is called the `HostLink`, since this is the link uptree, which indicates the path to the host system. Each of the other links of this node may have been used to boot another transputer. The transputers connected to such a link are represented as a branch in the boot-tree.

Note that interconnections may be available which are not represented in the boot-tree. Therefore, a value 0 in the `NBooted` array does not imply that this link is not connected. In fact, it may be connected to either the 'parent node' or to a transputer residing in another branch of the boot-tree. In either case, the link has not been used to boot another transputer from this node, so `NBooted` will be zero for that link.

Making software independent of hardware

The information concerning the network layout can be obtained with the runtime function `GetSysInfo()`. An example is given below:

Example .... Using the `SYSTEM` structure in a program

```
#include <stdio.h>
#include <system.h>
SYSTEM Sys;
int main()
{
   double Speed;
   GetSysInfo( &Sys );
   Speed = Sys.ProcessorSpeed / 4.0;
   printf("This transputer runs at %4.1f MHz\n", Speed);
}
```

The use of the unique identity number of each of the available nodes is exemplified below:

Example .... Using node identities (simple version)

```
#include <system.h>
int main()
{
   SYSTEM Sys;
   GetSysInfo( &Sys );
   switch( (Sys.Tn - 1) %3 )
   {
       case 0  :   Task_A();   break;
       case 1  :   Task_B();   break;
       case 2  :   Task_C();   break;
   }
}
```

In this example program, each node's identity number is used to divide three tasks equally over the available network. On a system of 3 transputers, each of the tasks will be executed by one node. When loading the same program into a network of 300 transputers, there will be 100 nodes working on each task.

In practice the first node often performs a supervising task and some message passing has to be done on each of the nodes. This could be added to the above structure by running a Message_Passer task on each node in parallel with the node-dependent calculator task, and starting a Controller task on the first processor:

Example .... Using node identities (extended version)

```
#include <system.h>
int main()
{
   SYSTEM Sys;
   GetSysInfo( &Sys );
```

```
        par {
            if( Sys.Tn == 1) Controller();
            switch( (Sys.Tn - 1) %3 ) {
                case 0 : Task_A(); break;
                case 1 : Task_B(); break;
                case 2 : Task_C(); break;
            }
            Message_Passer();
        }
    }
```

T4xx/T8xx transparency

When running a program on a network with both T4xx and T8xx transputers the same floating point calculations can be used on all processors. The floating point primitives are placed in resident libraries which are loaded when needed and according to the type of the transputer. The T4xx transputer will load `realt4.rsl` which contains emulations of the floating point primitives. The T8xx will load `realt8.rsl` with code using its floating point processor instructions. Both calculations are guaranteed to give the same results, but the T8xx is faster.

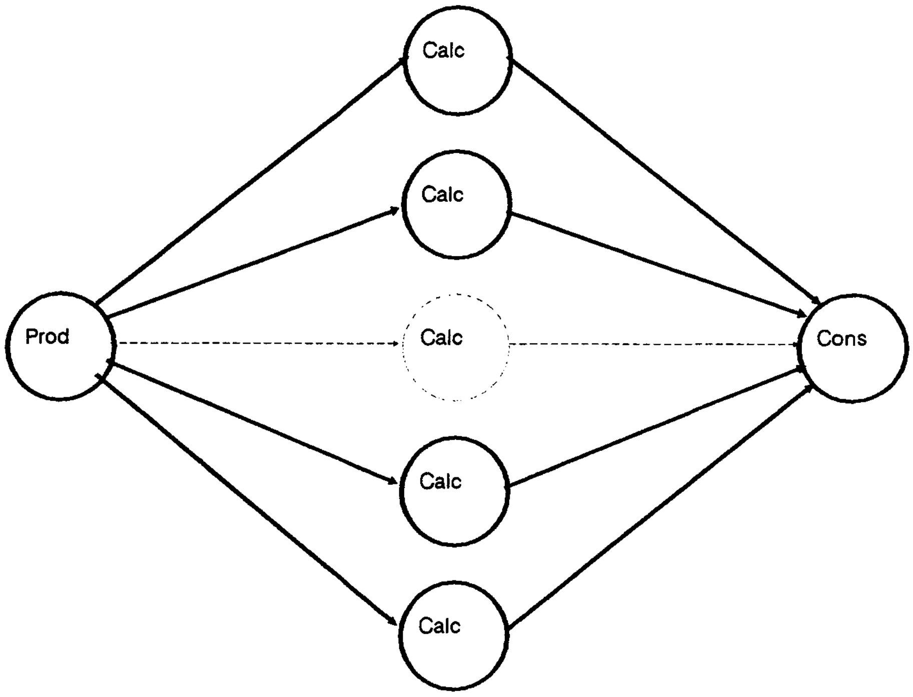
When this transparency for T4xx and T8xx transputers is not needed, the `#pragma fpu` preprocessor directive can be used on a T8xx to increase performance of floating point calculations by directly inserting the floating point instructions in the program code. When using math funcitions, linking `T8math.lib` instead of `math.lib` on a T8xx will increase performance considerably.

An example of optimising for floating point performance, while retaining the possibility of running the program on a mixed network, is given in "special topics" on optimisations.

The farmer demo program

Compute-intensive problems often contain parts in which the same algorithm is executed on different sets of data. In situations where the processing of each individual set of data is independent of the results of the same calculations on other sets, data concurrency can be used to speed up the overall performance of the program. Examples are ray-tracing, Monte-Carlo simulations and generation of images involving fractals.

The logical structure of such a program can be visualised as shown in the figure below: a Producer continuously produces command messages containing the data to be processed by one of the available Calculators. The result messages are sent to the Consumer, which collects and combines them in some way. This type of program is said to have a farmer structure: the Producer 'farms out' the work to a number of Calculators.

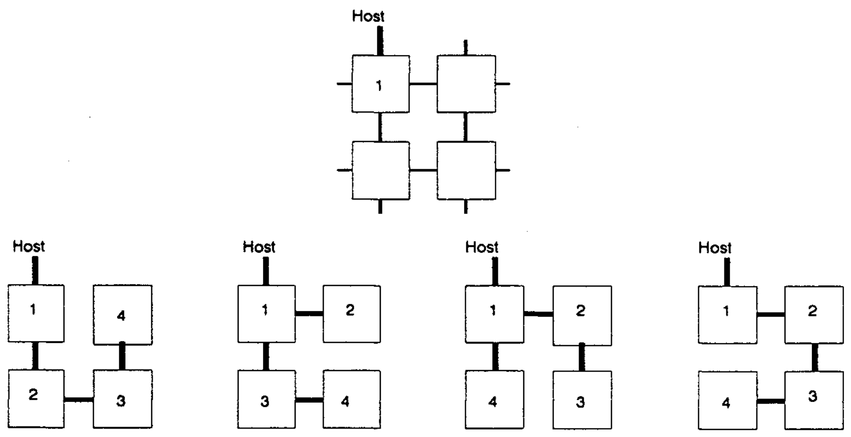


Logical structure of the farmer

Since the transputer has a limited number of links, this logical structure has to be mapped onto the physical structure of the available network. The scheme which is used to distribute command messages to the Calculators and to collect results, does not effect the logical structure of the program. However, the scheme highly influences the dynamic load balancing.

Implementation in the Par.C System

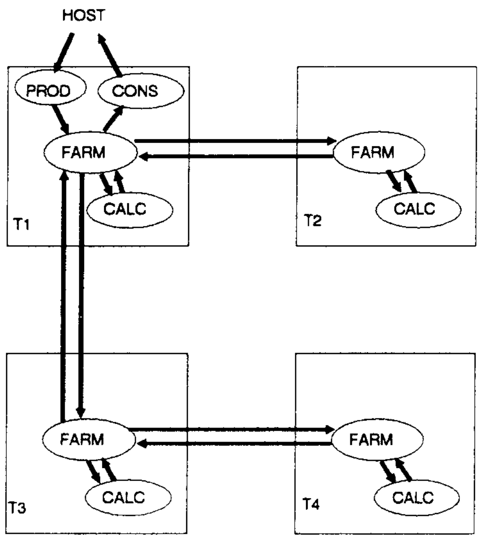
In the example program described in this paragraph, one possible scheme is implemented with a number of processes, collectively called the 'Farmer'. The Farmer demonstration program, which is included in the software of the Par.C System, makes use of the dynamic bootsystem, which generates a boot-tree describing the links through which the transputers are booted (see also one of the previous sections on the Par.C bootsystem). Information on the boot-tree is used to regulate the distribution of commands (travelling downtree) and collection of result messages (travelling uptree). An example of a four-node network with its possible boot-trees is shown in the figure below.



Possible boot-trees in a four-node network

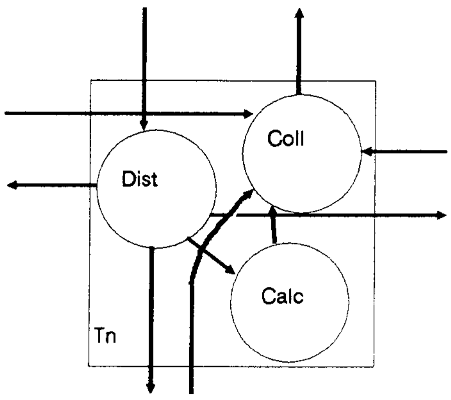
To make the logical structure of the program independent of the hardware, the Farmer processes are placed on every transputer, to handle all data transfer between the Calculators and either the Producer or the Consumer. Thus, the only processes talking to the links of the transputers are the Farmer processes. So the Producer, Calculator and Consumer processes only communicate with the Farmer processes.

The transputer's identity number (set by the bootsystem) is used to start the Producer and Consumer processes on the first transputer only. The Farmer and Calculator processes run on all transputers, including the first one. The Farmer processes running on different transputers are interconnected according to information about the boot-tree: each link which is used to boot a transputer through is also used to connect the Farmer process of that transputer to the Farmer process on its 'parent node' (see figure on farmer configuration). These connections are sufficient to reach all Calculator processes.



Example of farmer configuration on a four node network

The main tasks of the Farmer are the distribution of command messages, containing the data to be processed, and the collection of result messages, containing the resulting data. Since these tasks are independent of each other, separate processes for Distribute and Collect are started concurrently, as is shown in the following figure.



Overview of the farmer processes

The performance of the program is enhanced by using request buffer processes in the Distribute process: a request for the next set of data is sent by this buffer process, while the current set of data is still being processed by the Calculator. The Distribute process only accepts a new message coming down the boot-tree when it knows that it can propagate it to one of the request buffers. Together with the automatic synchronisation of the channel communication, this mechanism ensures that no processor will get overloaded with messages that cannot be handled.

The Collect process multiplexes the resulting messages coming from the Calculator and from the Farmers in the sub-trees. The buffers used for this process need only be simple buffers: no request for new data is needed.

Pseudo-code of the Farmer

The pseudo-code of the Distribute and Collect processes contained in the Farmer explains what happens in a more natural language than the Parallel C source code. The keyword `select` denotes a point where actions are only taken when one of the alternative events occurs (see section on select).

Pseudo-code .... Distribute process:

```
forever {
   select {
       alt guard request from calculator buffer on this node
               read request
               increment number of available buffers
       alt for all links
           if link is connected and downtree
           guard request from link buffer
               read request
               increment number of available buffers
       alt if at least one buffer is available
           guard message coming in from uptree link
               read this message
               if calculator buffer on this node is empty
                   decrement number of available buffers
                   send message to calculator buffer
               else
                   find an empty buffer to downtree link
                   decrement number of available buffers
                   send message to this link buffer
   }
}
```

Collect process :
```
forever {
   select {
       alt guard message from calculator on this node
               read this message
               set source_ID in message header
       alt guard message coming in from downtree link
               read this message
   }
   send message to uptree link
}
```

Running the farmer demo program

The farmer demo program can be executed by typing `run demo <total_time><job_time>`. The first parameter `total_time` can be used to determine the maximum time in seconds this demonstration program should be active. If this parameter is omitted, a default value of 10.0 seconds is taken. When this time has elapsed, the `exit()` function is called.

The second parameter `job_time` can be used to simulate the time in seconds the Calculator should take to 'process' the data before it can send back a result message. If `job_time` is omitted a default value of 1.0 will be used, meaning that the Calculator will be busy for one second from receiving the data to sending back the result. Note that if `job_time` is used, `total_time` must be specified also.

It is interesting to see what happens when the value of `job_time` changes. When using small values (in the range of 0.0 through 0.2), the ratio of calculation time over communication time is too low to keep more than a few Calculators busy, thus resulting in other transputers being idle. More of the available transputers will be put to work when `job_time` is increased.

Adaptations to the farmer structure

To this basic farmer structure, small changes can be made to enhance performance, or increase the functionality of the program. In the remainder of this chapter, a number of possible changes are discussed.

The source files of the Farmer demonstration program are included in the Par.C System software. If one or more of the following files are missing, you are kindly requested to contact your dealer for a new copy.

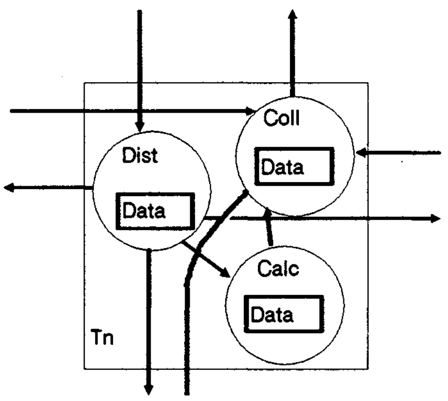
- `demo.c` : The user definable source code for the main, Producer, Consumer and Calculator functions.

- `farm.h` : The include file with definitions for the Farmer functions.
- `farm.c` : The source code for the Farmer functions.
- `demo.run` : The executable file that will run on any transputer network.

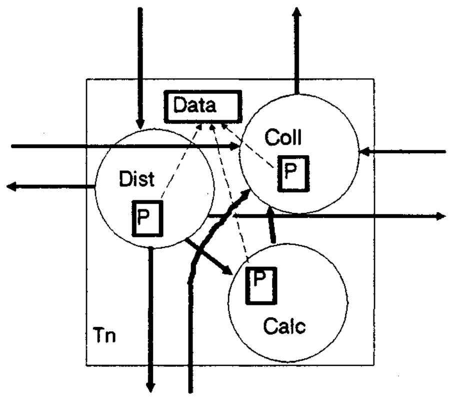
Producing a new message structure involves making changes to `farm.c` and to `farm.h`. Putting another calculator task into the existing farmer program requires a change of `demo.c`. If changes are made to the structures of messages from Producer to Calculator and back to the Consumer, `farm.c` needs recompilation as well, since the message structure typedef in `farm.h` is changed.

Optimising communication on the same node

The Farmer described earlier operates by communicating messages between the different parallel processes. The Farmer can be made more efficient by using pointers to the messages in communications on the same transputer, thus reducing the number of bytes that have to be communicated in these cases.



A)



B)

Passing data between processes Passing pointers inside one

The above figure illustrates the differences between passing data and passing pointers. In the first case, each process uses its own private memory to store the data. Message exchange between processes takes place by copying data from the memory of the sending process to the memory of the receiving process.

When messages contain a large amount of data, it may be more efficient to use the second setup, in which the data remain where they are, and pointers to the data are passed around between processes. Of course, since processes on different transputers do not share access to the same memory, data communication between these processes still has to be done by sending data over the links.

Note that simply adapting the Farmer processes to work with data pointers for internal communication is not sufficient to ensure correct functioning of the program. In the situation depicted in part A) of the figure above, the space occupied by a message is internal to each of the processes and therefore this space only needs to be allocated once inside the workspace of each process. Before a new message can arrive, the former one is copied to another process, and therefore the same memory space can be used without causing valuable data to be overwritten.

As mentioned earlier, in situation B) only pointers are passed around while the data remains where it was placed upon arrival in this particular processor. To avoid one message being overwritten before it is processed new space has to be allocated for each new message arriving from one of the links. Furthermore, to avoid memory congestion, this memory space has to be deallocated by the last process to handle each particular message, i.e. the process transferring the message to another transputer.

Including more statistics

The functionality of this Farmer can be extended in different directions by using a more complex message header. Including a `message_size` would provide the possibility of sending messages of different sizes. Other data can be included to obtain more statistics. For instance a `path_length` can be used to determine how many Farmers the message has passed, having each Farmer increment this number before passing the message. A `message_type` can be used to take different actions on different messages. This can be used for messages that have to be broadcast to all Calculators in the network.

One of the interesting statistics which can be obtained from a sophisticated header are timings of the system. By setting the starting time when a message is sent out by the Producer, the Consumer can determine how long it took to handle the message. The time spent in the Calculator can be determined also and used for analysis. These timings can be used by another program, for instance to determine the number of Calculators actually working, or the communication overhead in this particular system running this particular program. Some insight in the possibilities of dynamic load balancing can be obtained by changing the amount of work that has to be done to process each single command message.

Below the type definition of a more sophisticated message-header is given:

Example .... Extended message header

```
struct header {
   int      message_ID;
   int      source_ID;
   int      destination_ID;
   int      message_size;
   int      path_length;
   int      message_type;
   clock_t DT_Calculator;
   clock_t DT_Message;
}
```

`DT_Calculator` is used to store the time needed to calculate the result. `DT_Message` holds the time elapsed between producing the command message and consuming the result.

Placing different tasks on different nodes

As mentioned earlier, the tasks to be executed by the Calculator processes can be replaced with user-defined tasks by editing `demo.c`. It is possible to have different transputers execute different tasks, using the information received from the network analyser to decide what parts of code should be executed by which nodes.

To be able to distribute the work in such a program, the farmer structure has to be able to cope with command and result messages of different types. Therefore, the type information should be available in the Farmer processes. As a matter of fact, transputers executing different tasks can be seen as nodes of different types. These nodes are now combined into one tree, but as far as Farmer tasks are concerned, this single tree should be seen as a number of trees interleaved, one for each type of node.

Therefore, type information should be added to the command and result messages, and in the number of available buffers on each node: important information in this case, is how many buffers are available through each of the links, leading to a certain type of node. Note that the Producer and Consumer parts need to be changed as well.

Special topics

Program configurations

This chapter gives an overview of the different program configurations which are possible with the current version of the Par.C System.

In each of the sections of this chapter, an example is given on the basis of the same calculation task. In each of the programs, the program structure is the same: a Collector task collects data produced by one or more Workers. The placement of tasks and the code for the Workers differs from one example to the other, according to the configuration that it illustrates.

The PI example

The Calculator contained in the Worker code calculates the number PI, using the 'darts method', which is highly inefficient because it converges very slowly. Random points are chosen inside a square field, and a check is done on whether a point is inside or outside the largest circle which can be drawn inside the square. This could be viewed as throwing darts and counting the number of darts which have actually hit the board.

The approximation of PI follows from:

```
4*(number of hits inside circle)/(total number of throws)
```

Once in every timeperiod (specified in the code), each of the Workers in the system sends its results to the Collector, which adds everything up and displays the current approximation of PI on the screen, together with the number of darts that have been thrown sofar in the complete system.

To be able to reach the first Collector from the Calculators running on each of the nodes, the Worker code also contains a Message Passer. This process awaits messages coming from transputers down the boottree and forwards these in the direction of the Collector running on the first transputer.

Pseudo code for the Worker

In this chapter the exact code of each of the processes is not given, since the issue here is to show different configurations of Par.C programs. For the complete source of the following PI examples see the examples on the accompanying disks.

The pseudo code for the Worker part of each program is as follows:

Pseudo-code . . . . . Worker task

```
Transputer-dependent seeds for rand();
par {
   Calculator();
   Message_Passer();
}
```

Note that different seeds for the random number generator must be given to obtain a randomisation of the calculations over different transputers.

Organization of the source code

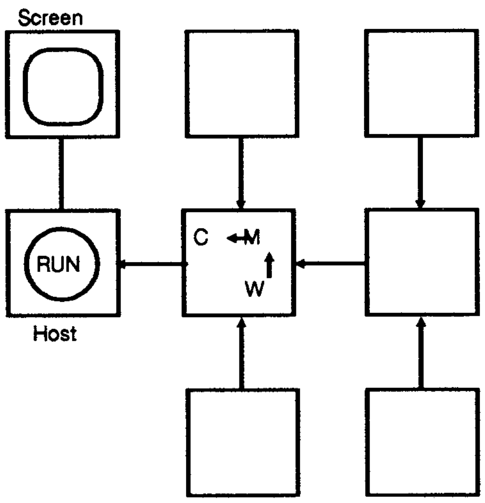
The source for the program on the example disk has been split up into several modules which can be compiled separately and then should be linked together. In this way the Collector has become a module, the Message Passer and the Calculator together have become a Worker module and the main function has become a module. As a result of this division of code, a new program configuration is obtained by compiling a new `main()` or `_entry()` function and relinking the modules. The possible link commands are described in the accompanying source files, and a batch file '`makerun.bat`' has been provided to produce a number of different versions automatically.

Single-node programs

When linking `a.lib`, only the first transputer will be booted, and only the characteristics of this first node will be checked and available through a call to `GetSysInfo()`. The status of all links except the bootlink is set to 'not connected'.

Example program: PI, (linked with `a.lib`)

When the PI program is linked with `a.lib`, only the first transputer is booted and executes one Worker task. At the same time, this first node contains the Collector task, which receives its data from the single Worker through a channel.



Single-node setup

Multi-node programs

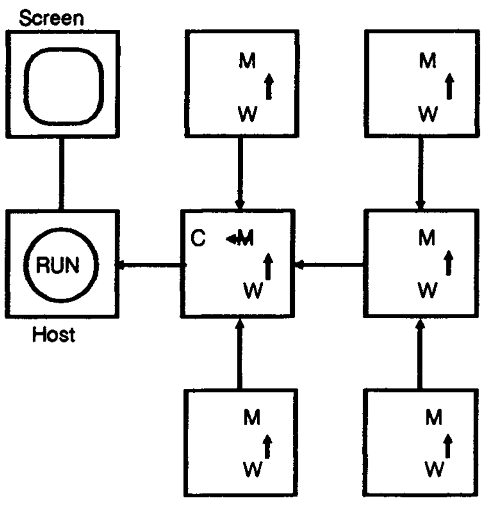
When linking `b.lib`, all transputers in the network are booted, and the code is copied to and started on all nodes. All information from the network analyser is available after calling `GetSysInfo()`. The copies of the program which are started on all of the nodes can determine which tasks to execute on the basis of this information.

Example program: PI, (linked with `b.lib`)

The only difference between PI running on one node and PI running on all nodes lies in the library which is linked: no other changes are necessary. When linked with `b.lib`, each processor in the network receives the code for the entire program. However, the Collector is only started on the first transputer, whereas the Worker tasks are started on all nodes.

Note .... A program linked with `b.lib` can also be run on a single-transputer network.

The description of the Farmer demo in the tutorial part of this manual gives a detailed discussion of a Farmer process running on a multi-node network.



Multi-node setup

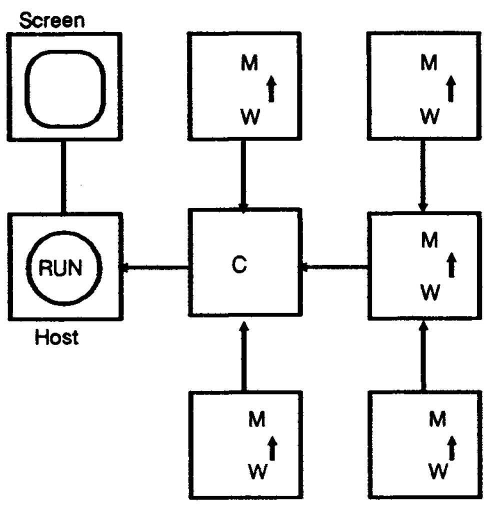
Subprograms for subsystems

The library function `Run()` can be used in combination with single- and multi-node programs to have a transputer boot its non-booted neighbour(s) with a separately produced executable program. `Run()` behaves like the Par.C Server which is running on the host system, with the linkadaptor address being changed to a link index to boot the neighbour through. The transputer(s) booted through the indicated link can be viewed as a subsystem. Note that the numbering of the transputers in this subsystem restarts at 1.

`Run()` can only be used when linking `run.lib`, and its usage is described in more detail in the library reference. The program being started from this 'parent node' must be linked with `noio.lib`, since `Run()` does not act as a file server.

Example program: RUN_PI

In this example, the Worker task is compiled separately and made into an executable program of its own. The Worker program is then loaded from the first transputer to a subsystem connected to one of its links, by calling `Run()`. The Collector task is also compiled as a separate complete program and is called RUN_PI.



Multi-node setup with Run()

The Worker program used with this example is PI_M, which also contains the necessary message passing code and runs on all processors in the subsystem. In the following section, this Worker will be explained in more detail and a version running on transputers without external memory will be described.

The advantage of using this configuration is that the Collector code is not copied to all processors (without being executed), since it is no longer a part of the main program containing the Worker task. This reduces the amount of memory needed by transputers in the subnetworks, especially when the Collector code is large.

Program loading and startup

Execution of a standard Par.C program in general can be divided into four phases:

1. Booting and investigating the available network
2. Loading the program into all available transputers
3. Executing the C startup code.
4. Executing the `main()` program

In the case of a single-transputer program (linked with `a.lib` instead of `b.lib`), phase 1 is replaced with a bootsequence for the first transputer only, and in phase 2 the code will only be loaded into that single transputer.

The memory occupied by the code used to execute phases 1 and 2 is almost completely released when phase 3 is entered. The only memory space remaining occupied contains the memory manager and some system information (about 512 bytes in total).

The loader always uses `_entry( sys_intern, CSEGstart, math )` as the entry point of the program. This is normally the entry point to the standard Par.C startup code, which in turn calls `main()`.

This is what the standard Par.C startup code does:

1. If needed, the resident part of `t8math.lib` is installed:

```
_InstallMathT8( math );
```

2. A number of global variables are initialised:

```
__CSEG__        _tohost
__DSEG__        _fromhost
__PROG__        _ttype
_Tn             _nT
```

3. The commandline is obtained from the bootlink, `argc` and `argv` are initialised, and the commandline is passed on to transputers which have been booted from this node.

4. If T4xx/T8xx transparent floating point arithmetic is needed, the resident libraries `realt4.rsl` and `realt8.rsl` are loaded from the bootlink and distributed to all nodes which are booted from this node. The library corresponding to the transputer type is kept in memory and references from the program to this library are resolved.

5. If the alarm function is included in the program code, a process for timer queue maintenance is started.

6. On the node connected to the host (`_Tn == 1`) the file I/O system is initialised by a call to `_finit()`, which will install the local fileserver as HOST0 and open the standard streams stdin, stdout and stderr. When `noio.lib` is linked, `_finit()` will only initialise the time facilities and the environment variables PATH, PARSEC, TIMEZONE and SUMMERTIME, and then call `Release()` to cause an exit of the server running on the host.

7. The user program is started by: `exit( main( argc, argv ) );`

Defining user supplied startup code

Defining `_entry()` in the user program will overrule the linking of the standard Par.C System startup code, and cause the loader to call the user program at `_entry()`. Note that in this case `main()` will not be called automatically.

The next example can be used as a frame for a user-supplied startup module. The differences with the standard code are :

- The variables are #defined instead of assigned.
- The variable `__PROG__` is omitted.
- The commandline is not parsed into arguments.
- The resident libraries are not installed.
- The timer queue is not initialised.
- `Release(0)` is called instead of `_finit()`.
- `main()` is not called, code just continues with user code.

Example .... Custom `_entry()`

```
#include <system.h> /* Description of system structure   */
#include <stdlib.h> /* Declaration of malloc(),realloc() */
#include <stddef.h> /* for definition of _Word and _Byte */
#include <string.h> /* for strchr()                      */

#define    _tohost     (sys.HostLinkOut)
#define    _fromhost   (sys.HostLinkIn)
#define    _ttype      (sys.ttype)
#define    _Tn         (sys.Tn)
#define    _nT         (sys.nT)

void       _InstallMathT8();
static     _Word   *GetDseg();
_Word      *__DSEG__;
_Byte      *__CSEG__;

#asm
GetDseg
   ldl  1
   ret
#endasm
```

```
void _entry( sys_intern, cseg, math )
_Word *sys_intern;
_Byte *cseg, *math;
{
   int link, psize;
   SYSTEM sys;
   char *cmdbuf = malloc(120), *argstart;

   __DSEG__ = GetDseg();     /* Get start of data segment  */
   __CSEG__ = cseg;          /* Get start of code segment  */
   GetSysInfo(&sys);         /* Get system info            */
   _InstallMathT8(math);     /* Install mathlib if needed  */
   if( _Tn == 1 ) getcmdline( cmdbuf );
       else _in( _fromhost, cmdbuf, 120 );

   par( link = 0; link < LinksPerTransputer; link++ ) {
       if( sys.NBooted[link] )
           _out( LINKOUT(link), cmdbuf, 120 );
   }

   realloc( cmdbuf, strlen(cmdbuf)+1);
   Release(0);               /* send exit command to server*/

#pragma fpu                  /* Enable in-line fpu code    */
   ...                       /* User program               */
   StopProcess();            /* Return is not possible     */
}
```

`Release(0)` sends an exit command to the server running on the host. From that point on, a user defined server protocol can take over if the user wants to start his own server on the host.

`StopProcess()` is used because `_entry()` is not allowed to return, as the caller (i.e. the loader) does not exist anymore after `_entry()` is called.

The resident libraries `realt4.rsl` and `realt8.rsl`, implementing T4xx/T8xx transparency for floating point arithmetic, are normally loaded by the standard Par.C startup code. When the resident libraries are not loaded, floating point arithmetic can only be used with the `#pragma fpu` compiler directive, and will therefore only be executable on T8xx transputers. Use the `_ttype` macro `(sys.ttype)` to avoid T8xx instructions being executed by T4xx type transputers. Note that the global variable `_ttype` is not automatically initialised.

If the program uses routines from the library `t8math.lib`, a small low-level routine must be installed. This is accomplished by calling `_InstallMathT8()` with the third parameter of `_entry()` as argument.

If `t8math.lib` is not used, it releases the memory occupied by this low-level routine, so it is always useful to call `_InstallMathT8()`.

The local file server can be installed by calling `_finit()` instead of `Release(0)`. However this will produce about the same amount of code as the standard startup code and is thus not recommended. When linking `noio.lib` it will supply support for time-related functions (i.e. `time()`, `localtime()` etc.) and restricted support for `getenv()`. The environment variables PATH, PARSEC, TIMEZONE and SUMMERTIME can be obtained.

`_finit()` may only be called from the node directly connected to the host, therefore time and environment variables can only be obtained on that transputer.

Building tiny programs

As is shown in the previous paragraph, the standard Par.C startup code contains a considerable amount of code to set up support of the various runtime facilities included in the Par.C System.

When producing tiny programs (for example when using transputers without external memory), this Par.C startup code can be minimised and tailored to support only those facilities which are used in the program. For instance, linking with `noio.lib` minimises the startup code for programs not needing file I/O functions.

The maximum size of a program which will fit into the internal memory of the transputer is somewhat smaller than the size of that internal memory: the loader for each transputer must also be present in order to load the program. The space occcupied by the loader is deallocated before `_entry()` is called, and can then be used by the program as workspace.

Minimising the memory needed for a program includes telling the linker to reserve very little space for the initial workspace : using -s0 will cause the initial workspace to be set at the default minimum stacksize. The default minimum workspace size is currently set to 40 words. In this case the program will probably run out of stack very soon, so that a new stack is allocated, using the now available space which was previously occupied by the loader.

The size of this extended stack can be influenced by the program by means of `_setFunstack()`. This kind of programming is balancing on the verge of an "out of memory" runtime error. If this error occurs, another value for the stacksize should be chosen.

A complete startup routine for a tiny program could look like this:

Example .... Minimising code using `_entry()`

```
#include    <system.h>
#include    <stddef.h>
#define     STACK    0x110    /* Needs experimenting */

void My_Main();

void        _InstallMathT8();

void        _entry( intern, cseg, math )
_Word  *intern,
_Byte  *cseg, *math;

{
   _InstallMathT8(math);    /* Install math.lib if needed */
   Release(0);              /* Send exit command to server */
   _SetFunStack(STACK);     /* set default size for new stack */
   My_Main();               /* force new stack frame */
   StopProcess();           /* termination of this program */
}
```

The program called could then look like this:

Example .... User program `My_Main`

```
#include <system.h>    /* Description of system structure */
#define     _tohost    (sys.HostLinkOut)
   ...                 /* Various definitions           */
My_Main()
{
   SYSTEM sys;
   GetSysInfo(&sys);          /* Get system info      */
   ...                        /* User program         */
}
```

Such a program can be loaded directly from the host, in which case a custom server must be written to take over communication with the program after the Par.C Server (RUN) exits.

Example program: PI_E

An example of a tiny program, which is started from the first node using `Run()`, is provided as PI_E. It differs from PI_M in that the latter uses the standard C startup code and calls `main()` from there, whereas PI_E is minimised with custom startup code.

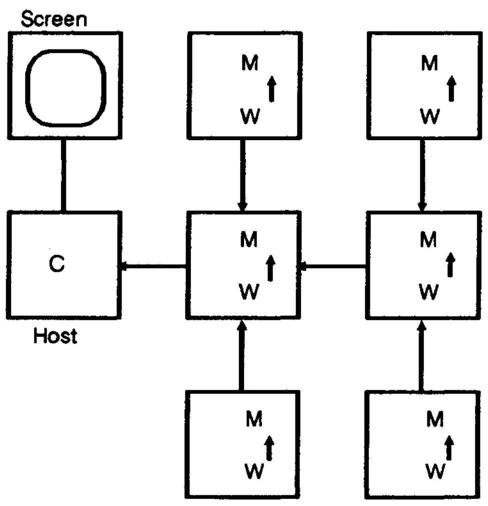
Custom server on the host system

It is possible to produce programs which can be downloaded and served without using the Par.C System loader/server. In this case, the started program must be structured according to the following rules:

- `_entry()` must be used to define custom C startup code.
- No file I/O may be used at anytime, since no Par.C System server is available.
- No environment variables, commandline, and time-related functions may be used, since these need the supporting file I/O system during startup time.
- The `Release()` function cannot be used, since there is no Par.C System server to send the exit command to.
- The `exit()` function cannot be used, since there is no return point to jump to. Instead, `stopProcess()` must be used to terminate.

Example program: PI_R

In this version of the PI-calculating example program, only the Worker task has been produced as a Par.C progam, and a special server is used from which the transputer network is booted. This server also executes the Collector task, which was previously done on the first transputer, using the Par.C System server to write results to the screen.



Multi-node setup with custom server

Memory usage in Par.C-compiled programs

Each process on the transputer has its own workspace, which is used as a stack. The figure below illustrates the structure of this stack.

For each concurrent process a new workspace is allocated, and the Bottom-of-Stack pointer, the Static Link and the Global pointer are initialized to the correct values. The Static Link is a pointer to the workspace of the process executing the par statement in which the process concerned is started (sometimes called "parent process").

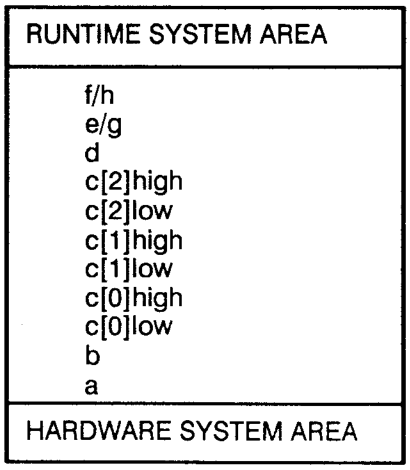
If the available memory is insufficient to start a process, a runtime error will occur, causing the program to exit.

The runtime system area contains pointers to the global data area and to the parent of the current process. The locations with small negative offsets measured from the workspace pointer are reserved for system use in the transputer's hardware. These locations do not carry lasting information, but must always be allocated directly under the current workspace pointer.

During execution of a select statement, a small number of locations in the upmost area of the local space are reserved for system use. In these locations, the status of the process executing the select is stored, and some temporaries are allocated when setting up the guards specified in the alternative clauses.

Example .... Allocation of local variables

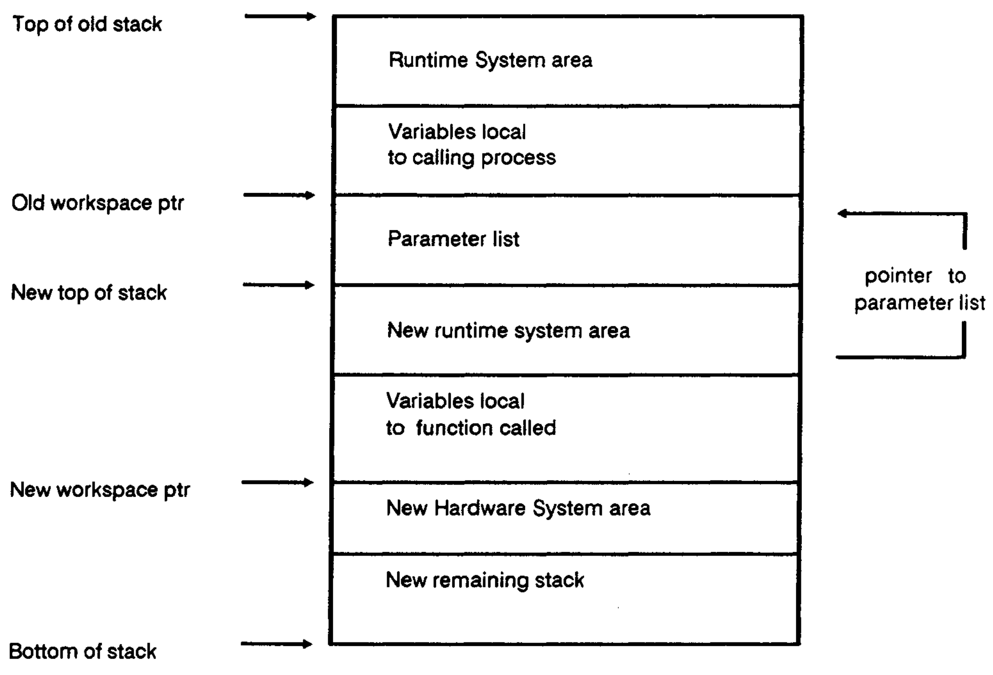
```
{  int a,b;
   long c[3];
   int d;

   if (condition){
       int e,f;
       ...
   }else{
       int g,h;
       ...
   }
}
```



The local variables are allocated in order of declaration. When locals are declared in compound statements inside the process, these are therefore allocated above the locals declared on a higher level. Since the instruction length on the transputer depends on the size of the operands, the locals nearest to the workspace pointer are fastest accessible. This point is worth noticing when declaring large arrays.

During evaluation of complex expressions, more stack may be needed to store temporary variables. The code generated by the compiler guarantees an amount of remaining stack which is sufficient to store all necessary temporary variables. The extra memory needed is allocated by lowering the workspace pointer. After completion of the expression evaluation, the workspace pointer is readjusted to its original position.

When calling a function, parameters are placed in a list and a pointer to this parameterlist is stored in a new runtime system area. The workspace needed for execution of the function is compared with the size of the remaining stack. The remaining stack is used when its size is sufficient for execution of the called function. Otherwise, a new stack is allocated automatically. The situation after calling the function is shown in the following figure.



Situation after calling a function

Methods for speed improvement

The Par.C System offers many possibilities to optimise performance. From the source level down to the linking stage the user can influence the final result.

- On the level of program structure, the user can make use of the knowledge of the target transputer type: floating point calculations will be executed much faster when run on a transputer with on-chip floating point unit.
- On the source code level the choice of the right data type for variables can cause considerable improvements.
- To optimise for the size of the program, the assembler can be instructed to adjust the length of external references.
- At link time, performance can be enhanced by optimal linking order, and intelligent use of internal (fast!) memory.

Since the effects of these optimisations are application dependent, the user should have knowledge of when and how to use them. In the following paragraphs we try to provide some of this knowledge.

Optimisations in program structure

An important source for optimisations of performance lies in the structure of the parallel program that is produced. Using buffer processes to keep calculating processes from having to wait for data is one example.

However, this is not a subject which can be discussed at length in this manual. In this chapter, we will indicate a limited number of optimisation techniques on different levels.

Source level optimisations

- Use register storage class when declaring function parameters which are referenced more than twice. This will create a local copy of the parameter, instead of accessing it via the pointer to the argument list.
- Declare heavily used variables first. If it is your style to use variables with limited scope, for instance within loops, declare them with register storage class. This will instruct the compiler to allocate them in a low offset area, reducing instruction size.

- Do not use shorts unless you really have serious memory shortage. In contrast to int or char types, short integers are not supported by the instruction set of 32 bit transputers. 16 bit memory references have to be simulated by byte oriented operations, which makes them very inefficient. If you have to use shorts, use them as little as possible in expressions.
- Normal integers are 32 bit, longs are 64 bit. Be sure not to use longs if you only need 32 bits, as this will decrease performance.

Assembler optimisations

Transputer instructions and operands are coded into a variable number of bytes of object code, depending on the number of prefixes needed to encode each particular instruction or operand. In the case of instructions, nothing can be changed in this: their code is fixed. Instruction operands are coded in as few bytes as possible by the assembler.

Optimisation can be performed on the length of references which are used as operands for load, store, jump and call instructions. On a 32-bit transputer, the operands can take anything from 1 to 8 bytes, depending on the offset of the reference. The assembler optimises the instruction length as much as possible, but it can not determine how large an external reference must be, as these will be filled in by the linker. Therefore, the assembler uses fixed default lengths for external references. These are 6 bytes for data references and 5 bytes for code references. These values will usually be sufficient. However, in many cases these lengths will be too large, resulting in unnecessary "pfix 0" instructions.

There are two assembler commandline options which influence the number of bytes reserved for external references. These are "-ic" and "-id" for code and data reference lengths respectively. For example, using "-ic 3" will cause the assembler to reserve 3 bytes for any call or jump to an external codelabel. The best way to determine whether unnecessary space was reserved is to inspect the map file which can be produced by the linker. The linker must be invoked with the "-m" option to obtain this file. For every object file, the column '`Req/Act`' gives infomation about the requested and the actual instruction length of external data and code references. If the requested length is larger than the actual length, reassemble the file with the "-ic" or "-id" switches indicating the appropriate values.

Linker Optimisations

The linker can be instructed to place parts of the program in the fast internal memory of the transputer by the command line switches "-bs", "-bd" and "-bc", which are discussed below.

-bs

This places the initial stack in internal memory. Note that this is only the stack of the main program. Any concurrent process started in a `par` construct or using `RunProcess()` will get a stack elsewhere. Also, if the program should run out of stack, the newly allocated stack will reside elsewhere. The linker switch "-s <n>" can be used to enlarge the initial stacksize to <n> bytes. Using "-bs" can be advantageous if functions are deeply nested, for instance when using recursive functions. Also, programs which mainly reference local data (which are always located on the stack) can be speeded up considerably using this switch.

-bd

This places the data segment in internal memory. This can be advantageous in programs frequently referencing global data.

-bc

This places the code segment in internal memory. This method can best be used if a program does not reference much data, for instance if the program does a lot of processing using constants. It can also be used if the data segment is large and the code segment is relatively small. Since only a part of the data will fit in internal memory, only references to that part will contribute to an increase in speed.

If you use any of these switches, a little attention to the order in which you link the modules often pays off. The linker allocates data and code blocks in the order in which the corresponding object files and libraries appear on the commandline. If either the code or the data segment is placed low, linking the most frequently used modules first will render the largest effect.

Using the floating point unit efficiently

If your transputer network only contains T8xxs, you might wish to take more advantage of the floating point unit. Here are a few suggestions.

- Use of the compiler directive `#pragma fpu` in Par.C source files.

The compiler will emit fpu instructions instead of generating calls to the resident libraries to handle the elementary floating point operations.

- When using math functions, `t8math.lib` should be linked instead of `math.lib`.

This library uses the T8XX fpu instructions directly rather than via calls to `realt8.rsl`. This has made it possible to interleave code for the cpu and fpu as much as possible, thereby using the on-chip parallelism of the T8XX transputer in a much more effective way. In most cases, performance will increase dramatically. Even if the network contains T4XX transputers, `t8math.lib` and `#pragma fpu` might be used. However, the user should take care that the code involved will not be executed if the program runs on a T4XX. This can be accomplished by making execution of floating point arithmetic dependent of the result of a test for the transputer type.

Example .... Optimising floating-point performance

```
main()
{
   par {
        if(_ttype==8) DoFloats();
        PassMessages();
   }
}

void PassMessages();
{
   ...
}

#pragma fpu
void DoFloats();
{
   ...
}
```

In this example, an outline is given for a program running on transputer networks which contain both T8XX and T4XX transputers. The latter processor type will only have to pass messages from the host system to the T8XX nodes and back. Since `DoFloats()` is only started on T8XX nodes, the program as a whole will run on any transputer network. Note that the code for the floating point calculations will also be loaded into the T4XX nodes, but will never be executed there.

Combined use of `#pragma fpu` and linking `t8math.lib` will cause both the floating point primitives (like add, multiply, etc) and the high level functions (like `sin()` and `cos()`) to be speeded up.

Inline assembly

Inline assembly can be used through the preprocessor directives `#asm` and `#endasm`. This section indicates a few points which should be kept in mind when using this facility.

Usage

The preprocessor directive `#asm` indicates the start of the inline assembly. Conversely, `#endasm` indicates the end. The lines between `#asm` and `#endasm` will be copied directly from the source file to the assembler file. Due to this compiler bypass, the compiler will not update its symbol table with the identifiers named in the assembly. The compiler does not know about the types of functions written using inline assembly either, so a declaration of these functions should be made at C-source level. One more side effect of the bypass is that the parser may not be ready processing the preceding C tokens when the inline assembly is copied to the assembler file. This can result in assembly ending up in the wrong place. To avoid this, inline assembly should be inserted only at the following points in a C source file:

- Following a declaration without initializer. This can be used at toplevel, i.e. outside function definitions.
- Following an empty statement ("`;`"). This can be used inside function definitions.

Format

Assembly source code should be written according to the following rules:

- labelnames are placed at the beginning of the line, and are significant up to 31 characters.
- instruction mnemonics are indented to distinguish them from labels.
- operands follow the instruction mnemonic on the same line.
- any characters following a semicolon and up to the next newline are treated as comments. The semicolon introducing a comment may also be placed on the first position of a line.
- Empty lines are ignored.

Reserved labels

The compiler generates assembler labels consisting of a dot followed by a number (e.g. ".137"). If you use labels of this form in your inline assembly, you might get an error message from the assembler, indicating a multiply defined label.

Assembler code

To make a defined code label extern, you should use the `.public` directive followed by the code label:

Example .... Defining an external label

```
   .public MyAsmFunc
MyAsmFunc
   stl 0
   ...
```

If you want to reference an external code label, for instance to call a library runtime function, you should include a `.public` directive as well:

Example .... Referencing an external code label

```
   .public clock
   call    clock
```

Assembler data

For external data references there should be a `.global` directive:

Example .... Referencing an external data label

```
   .global     name,4            ;name and size of object
   ...
   ldl     ws+GP                 ;load global pointer
   ldnl    #name                 ;load value of 'name'
```

The '#' preceding 'name' is an operator which converts a byte offset to a word offset. This is necessary, because assembler labels are always treated as byte offsets, whereas the instructions `ldnl`, `ldnlp` etc. are word addressed.

To define storage in the data segment, the directives `.dseg`, `.word`, `.byte` and `.public` should be used:

```
.dseg
```

This directive switches to the data segment. The assembler always starts in the code segment. Switching back to the code segment is performed by the directive `.cseg`

```
.word <value1>,<value2>,...
```

This directive defines words in the data segment.

```
.byte <value1>,<value2>,...
```

This directive defines bytes in the data segments. Values can be either numbers or double quoted strings. Note however, that strings will not be null terminated automatically.

```
.public <name>,<size>
```

This directive makes the symbol < name > external, and associates a storage of < size > bytes with it. A label < name > should be present in the data segment.

Example .... Defining storage in the data segment

```
   .dseg
   .public     Glob,4
Glob
   .word       0
```

C calling interface

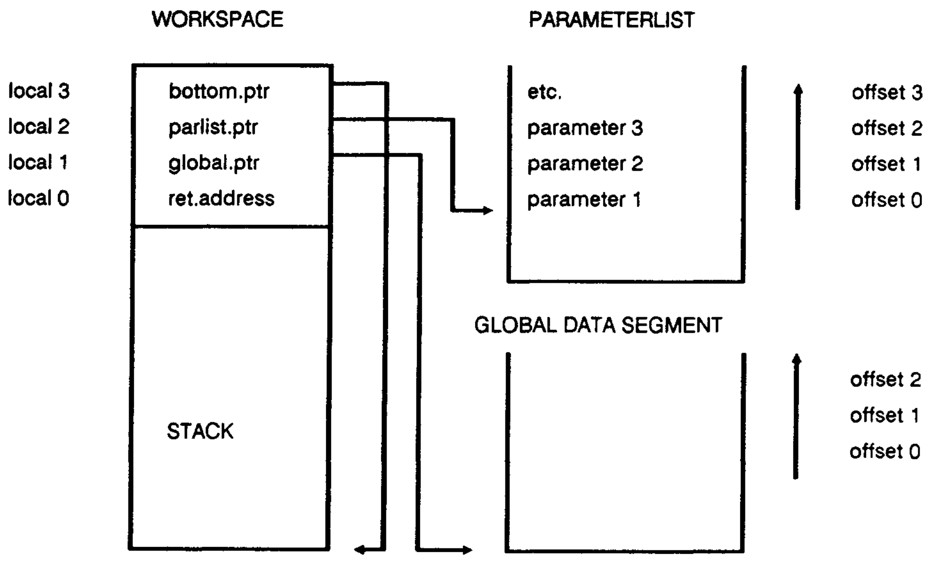
A call to a C function looks like this:

- Generate the argument list, the first argument lowest in memory
- Load the pointer to the bottom of stack (BOS)
- Load a pointer to the first argument (AP)
- Load the pointer to the global data area (GP)

- call the function.

BOS and GP can be found in the top of the workspace of any function called using the C interface. At function entry, BOS can be found at < Wptr > + 3 and GP at < Wptr > + 1. The arguments of the function can be accessed via the argument pointer, which is located at < Wptr > + 2 at function entry. The return value of the function is returned in Areg in case of a one word return value and in Areg and Breg, with Areg holding the least significant word, in case of a two word return value (longs, doubles).

To make sure that the return address will be saved, a '`stl 0`' instruction should be issued directly at function entry. This is because the '`gcall`' instruction, used for indirect function calls, does not store the return address automatically. The following diagram shows the layout of the stack directly after a function call.



Situation after a function call

Example .... Inline assembly function calling a C function

```
extern int f();     /* declaration info for the compiler */
#asm
;assembly for:
;int f(x)
;int x;
;{
;  return g(x,1);
;}
```

```
        .public f           ;defined here
        .public g           ;referenced here

ws      .equu   2       ;workspace for this function
x       .equ    0       ;offset of x in the argument list
BOS     .equ    3       ;offset of bottom of stack pointer
AP      .equ    2       ;offset of the argument pointer
GP      .equ    1       ;offset of the global pointer

f
   stl      0           ;save return address
   ajw      -ws         ;create workspace
   ldl      AP+ws       ;load argument pointer
   ldnl     x           ;load x
   stl      0           ;first parameter for g
   ldc      1           ;load 1
   stl      1           ;second parameter for g
   ldl      BOS+ws      ;load bottom of stack pointer
   ldlp     0           ;load pointer to argument list
   ldl      GP+ws       ;load pointer to global pointer
   call     g           ;call function, result is in Areg
   ajw      ws          ;deallocate workspace
   ret                  ;return from function
#endasm
```

Stack usage

The stack checking mechanism of the Par.C System ensures that 64 words of workspace are available when entering a function from C-level. For small, non-recursive functions this is often quite enough. If more stack is needed, the stack checker should be invoked. This will allocate more stack if the requested amount plus the 64 extra words exceeds the remaining stack. An entry to a recursive function using 10 words of workspace per call looks like this:

Example .... Checking stack usage

```
   .public     f
   .public     .checkstack
f
   stl     0
   ldc     10                ;load the number of words needed
   call    .checkstack       ;call the stack checker
   ...
```

The Par.C compiler uses a version of the stack checker which is optimized for speed. If you are interested in this, take a look in a compiler produced assembly file. (For ease of reading, compile with the `-ts` option. The assembly output will then be commented with the source code.)

Final remarks

It is difficult to use inline assembly inside a function body, because the exact offsets of local variables are unknown in most cases. Most common use of inline assembly is therefore to code complete functions, which can be called from C to perform a certain task.

You might consider writing complete assembler files, thereby skipping the compiler phase. These files can be assembled and linked just like compiler produced assembly files.

Utilities

SYSNET

A utility to show the available tranputer network

Description ..... The SYSNET utility is a multi-transputer Par.C program which shows information about the transputer network on which it is run. Apart from being useful while installing or reconfiguring your transputer system, the SYSNET program is also instructive for using the Par.C System. The source of this program is part of this package, and can be found in the EXAMPLES directory.

Usage ....

```
run sysnet
```

Before a Par.C program is started, the runtime system supplies each transputer in the network with information about the connections of its links, the available memory, the type of the transputer etc, in the form of the SYSTEM structure. One component of this structure indicates the link from which the transputer was booted. The 'bootlinks' of all the transputers in the network collectively define the 'boottree', and following the branches of this boottree towards the root of the tree renders the 'bootpath' of a transputer. In the SYSNET program, each transputer sends its SYSTEM structure via the bootpath to the root transputer (T1). T1 prints all the incoming information on the screen in a diagram showing most of the important network parameters.

Figure .... Possible SYSNET output

```
Number of transputers in system: 4
Ident   Type  Mhz    Link 0  Link 1  Link 2  Link 3   MemTop
   1    T800  17       -     * Host  <- 2.0  <- 4.2  80001000
   2    T414  20     * 1.2      -    <- 3.2     4.0  80040800
   3    T414  20       -        -     * 2.2     4.1  80100800
   4    T414  15       2.3     3.3    * 1.3      -   80100800
```

The four link columns indicate the node number and link number to which they are connected. Each transputer has one link marked with an asterisk, indicating the link over which it has been booted (i.e. the link to its 'parent node'). The arrow indicates a link which has been used to boot a 'child node'. The amount of memory per transputer can be read from MemTop by subtracting MemBot (= MinInt = 80000000), and is indicated in hexadecimal notation. The identity numbers as given in the first column, are known to the transputers themselves and can therefore be used in a Par.C program to influence its execution.

For more information about the SYSTEM structure, see the paragraph about the 'Network analyser' and the header file `system.h`.

Pseudo code ..... The SYSNET program is written using the following algorithm:

```
if I am T1
   print my information;
else
   send my information uptree;

for every downtree link
   for every node connected downtree
       receive information from this link;
       if I am T1
           print this information;
       else
           send this information uptree;
```

In this manual, the direction of the host system is referred to as 'uptree' and the direction towards the 'leaves' of the boottree is indicated as 'downtree'.

Note that this program does not examine the network itself, but that the runtime system has already done this for you. This means that the information available in the SYSTEM structure can be used in a similar way in other applications, to have the program adapt itself to the situation during runtime.

See also .... Tutorial, paragraph on dynamic parallel processing.

RUN2EXE

A utility to create stand alone programs

Description ..... The RUN2EXE Utility combines a transputer executable file (`.run` file) with a special version of the Par.C loader/server into a single executable file for the host (`.exe` file). Executing this program will boot the transputer network, download the transputer code and start the included server, just as if `run program` had been typed.

Usage .... The RUN2EXE utility can be invoked with:

```
run2exe [-o <outfile>] <infile> [server options]
```

`<infile>` is the name of the transputer executable (`.run`) file.

RUN2EXE itself has only one option:

```
-o <outfile>
```

This will make `<outfile>` the name for the resulting executable file.

In a stand alone application it is not possible to pass any options to the included server, because any additional commandline arguments are passed to the transputer program. To compensate for this inconvenience RUN2EXE recognises most of the options accepted by the Par.C loader/server and stores them in the resulting executable file, so that they can be used by the special included server. This means that the server will then always use the specified options.

The possible server options are:

`-p<number>`	: set link adaptor port address to `<number>`
`-r`	: do not reset before loading program
`-t`	: display execution time of loaded program
`-b`	: open all files in binary mode
`-x`	: load program and terminate without serving I/O requests

Notes .... Although a special version of the Par.C loader/server is used, it still needs access to the files `hardware.cnf`, `realt8.rsl` and `realt4.rsl`. If the file `hardware.cnf` is not available, the server uses a default setup. The files `realt4.rsl` and `realt8.rsl` can only be omitted if the Par.C program does not use any floating point operations or if the program is compiled with `#pragma fpu` to generate inline floating point code.

Error messages

RUN2EXE may generate the following errors:

`out of memory`

RUN2EXE could not allocate enough memory.

`file error in <name>`

An error occurred while accessing the file `<name>`.

`CRC error`

RUN2EXE's integrity is violated.

`file <name> is not available`

The source file `<name>` could not be found.

`missing input filename`

The source filename was not specified on the command line.

`unable to open file <name>`

The output file `<name>` could not be opened for writing.

See also: .... Par.C loader/server.

Error messages of standard options routine

`#pragma fpu`

`realt4.rsl`

`realt8.rsl`

`hardware.cnf`

MODRUN

A utility to modify Par.C executables.

With MODRUN the placement of code, data and initial stack low or high in memory can be controlled, the stacksize can be set/modified, the top of memory to be used by the module can be set (from version 1.22 upwards), and the program can be modified to run in either single- or multi transputer mode.

As MODRUN does this much faster than re-linking the program, it can be used when experimenting with different stack sizes, and locating different program segments in high or low memory (e.g. to optimise performance of the program).

Usage .... `modrun <filename> [options]`

If no extension is supplied for the filename, `.run` is assumed. A summary of options and syntax can be obtained by invoking MODRUN without arguments. These options are described in more detail below.

If only a filename is supplied, the file is analysed and a description concerning the sizes of the segments in the file is displayed, in the same way as the linker does after succesfully producing an executable file. The sizes are given in hexadecimal notation.

Example .... Possible output when invoking MODRUN with only a filename:

```
Booter :   bf5         Stack  :  400 (low)
Code   :   355c        Entry  :  2c8
Data   :   22c (low)   Udata  :  44
```

The segments marked `(low)` are indicated to be placed in low memory.

The options `-b`, `-h`, `-m`, `-s` and `-c` may be used to have the loader parameters modified. Before applying the modifications, MODRUN asks for confirmation:

```
Modify module.run (y/n) ? :
```

This can be answered with "Y" or "y" followed by [RETURN] to confirm. The question can be suppressed by using the `-nq` (for "no questions") option.

After adjustments have been made MODRUN will display :

```
Program modified successfully
```

Note .... The version of MODRUN should be the same as the version of the Par.C System with which the `.run` file was created. If not, the result is undefined.

Options

Setting the initial stacksize

The option `-s<size>` causes the initial stack size to be set to `<size>`. If the initial stack is chosen too large it is a waste of memory, but if the stack is small, a new stack must be allocated more often. This may cause a considerable degradation of performance. Tuning the stacksize to an optimum needs some experimenting, but often pays off in gained performance. See the Special Topics chapter, paragraph on speed improvement for more details on the possible results. The wanted initial stacksize can be specified according to the following format rules (which may be combined in any way):

- A plain number is taken to be the wanted size in bytes, in decimal notation.
- A number preceded by '`0x`' indicates that the size is given in hexadecimal notation.
- A number preceded by '`0`' indicates that the size is given in octal notation.
- A number preceded by a plus sign indicates that the current size should be incremented by the specified amount.
- A number preceded by a minus sign indicates that the current size should be decremented by the specified amount.

Examples .... Some possibilities in changing the stack size of a program:

`-s100`	sets the initial stack size to 100 bytes.
`-s0x100`	sets the initial stack size to 256 bytes.
`-s+100`	adds 100 bytes to the current size.
`-s-0x10`	subtracts 16 bytes from the current stack size.
`-s0100`	sets the initial stack size to 64 bytes.

Placement of code, data and stack segments

The -b and -h options can be used to influence the placement of code, data and/or stack segments in low (-b) or high (-h) memory. If more than one segment is to be loaded in low memory, a fixed sequence is used: first the code segment, then the (initial) stack, and finally the data segment.

Note .... Placement of code, stack and/or data in low memory is mainly used for optimisation of performance of a program: the on-chip memory of the transputer is normally much faster than external memory. MODRUN will not give a warning if the total size of segments which are to be placed low in memory exceeds the size of the internal memory of the transputer: a part of the segments will simply cross over the boundary between internal and external memory, since this boundary does not exist for the memory manager.

Examples .... Some possibilities in placement of program segments:

-bc	causes the code segment to be loaded low in memory.
-bsd	causes the stack & data segments to be loaded low in memory.
-bds	has the same effect as -bsd

If either of the -h or -b options is specified, MODRUN will display the layout of the loadmodule as if the modifications have been made already. Then it will ask for confirmation of the changes.

Setting the memory size on T #1

The -m option enables the setting of a memory size on the first node in the network. This will keep the network analyser on this first processor from performing a memory check, and instead assume the memory size to be as indicated after the -m option.

The -m option has been provided for specific interface boards, which have video RAM or memory-mapped I/O immediately following the normally usable RAM. Setting the memory size to an address below the actual size will prevent loading the program in video memory in the first case, and prevent a possible host system crash (resulting from an illegal or incomplete I/O access) in the latter situation.

The rules for specification of the memory size are the following:

- A plain number is taken to be the size in bytes, given in decimal notation.
- A number preceded by '0x' is taken to be the size in bytes, given in hexadecimal notation.

- A number preceded by '`0`' is taken to be the size in bytes, given in octal notation.
- A number followed by the letter '`k`' (or '`K`') is taken to be the size in Kilobytes: the number is multiplied by 1024 (0x400).
- A number followed by the letter '`m`' (or '`M`') is taken to be the size in Megabytes: the number is multiplied by 1024*1024 (0x100000).
- A number followed by the letter '`g`' (or '`G`') is taken to be the size in Gigabytes: the number is multiplied by 1024*1024*1024 (0x40000000).
- The '+' and '-' (binary) operators may be used.
- The number `0` causes the automatic memory check to be restored.

Note .... MinInt (0x80000000) is added to the specified memory size to obtain an address for MemTop. The maximum size that can be set with MODRUN is 2 Gigabytes (with MemTop set to the address 0x00000000).

Examples: ....

Option	MemTop at	Memory size
`-m2048`	0x80000800	2 KBytes
`-m0x1000`	0x80001000	4 KBytes
`-m4k`	0x80001000	4 Kbytes
`-m1m+2K`	0x80100800	1026 KBytes
`-m4M-64K`	0x803f0000	4032 KBytes
`-m1G+1M`	0xC0100000	1025 Mbytes
`-m0`	According to results of memory check	

Single-node or all-node program

The option `-cs` disables network investigation and causes the program to run only on the first transputer. Relinking the program with `a.lib` has the same effect. After confirmation MODRUN will display the message:

```
Making module.run a single-transputer program
```

The option `-cm` enables network investigation. The program will then be 'floodfilled' on all available nodes. Relinking the program with `b.lib` will have the same effect. After confirmation MODRUN will display the message:

```
Making module.run a multi-transputer program
```

Note .... The `-cs` and `-cm` options only work for programs produced with the Par.C System version 1.3 and higher.

Control of screen output

The `-nq` option suppresses the question for confirmation, which is normally prompted before the loadmodule is modified.

The `-ll` option causes library and copyright information to be printed to the screen, if this information is available in the load module that is being examined.

The `-ln` option suppresses screen output of MODRUN. This may be useful when calling the utility from a batch file. Note that using this option does not overrule the confirmation question. To have MODRUN do its job completely in silence, both `-ln` and `-nq` options must be used.

Error messages

There are 2 types of errors: syntax errors and processing errors. With one exception, all errors are fatal, causing the module not to be modified. The only non-fatal error is :

```
MODRUN - *ERROR* Reference patch segment misaligned
1 byte
```

This error is caused by an internal error in the Par.C System linker. If this occurs, you are kindly requested to fill out the bug report form and return it to Parsec Developments.

Syntax errors

Syntax errors refer to the commandline, and have the following format:

```
Syntax error: <description> [<info>]
```

The syntax errors are numerous and completely self-descriptive and will therefore not be discussed in detail.

Processing errors

Processing errors have the following format:

```
MODRUN - *ERROR* <description> [<info>]
```

A list of all possible fatal processing errors is given below. All numbers in the messages below are used as an example. They might have other values in an actual message. Wherever < name > is used, it denotes the name of the module as given in the commandline. The names '1.tmp ' and '2.tmp' denote temporary files as generated by MODRUN on the host system.

```
MODRUN - *ERROR* can not open <name>
```

The module to be modified could not be opened. Probably the path or file specification was incorrect, so an attempt was made to open a non-existing file.

```
MODRUN - *ERROR* write error, file might be corrupted
```

This is the only case, where MODRUN might have corrupted your module. It is advisable to relink your program, and also to check your disk system.

```
MODRUN - *ERROR* Can not change single/multi mode :
Failed to open temporary file
```

A temporary file is used to compose a new version of the load module. In this case it could not be opened.

```
MODRUN - *ERROR* Read of <name> failed
```

A read error occured while copying part of the module to the temporary file.

```
MODRUN - *ERROR* Write to temporary file failed
```

A write error occured while copying part of the module to the temporary file.

```
MODRUN - *ERROR* Can not change single/multi mode :
Failed to rename <name> to 2.tmp
```

After creating the new module in a temporary file, the original file is renamed. This failed. Try using MODRUN from the directory which contains the loadmodule.

```
MODRUN - *ERROR* Can not change single/multi mode :
Failed to rename 1.tmp to <name>
```

After giving a temporary name to the old module, the new module is renamed to the original name. This failed.

```
MODRUN - *ERROR* Can not set memory top : no booter in
module
```

A memory size is specified, which cannot be patched in the module since it does not contain bootcode.

```
MODRUN - *ERROR* Can not set memory top : not available
```

The module is made with a version of the Par.C System older than 1.22, which does not allow the possibility of setting the memory size. Relink the module with the Par.C System version 1.22 or higher.

```
MODRUN - *ERROR* Can not change single/multi mode :
version < 1.3
```

The -cs or -cm switch can not be applied on modules produced with the Par.C System older than version 1.3.

```
MODRUN - *ERROR* Data section has odd size : 7f2
```

The data section size should be a multiple of 4. This does not mean that the loadmodule is corrupted: it is possible to create a module with odd data size by (ab)using assembly source code. As this can cause unpredictable behaviour at loadtime and during execution, we recommended to scan any assembler modules incorporated in the program to contain a misaligned directive in .dseg. In most cases the linker will have given the warning:

```
*WARNING* Data label LABELNAME at odd boundary
MODRUN-*ERROR* Initialised data section has odd size: 883
```

The same story applies as for the previous message.

```
MODRUN - *ERROR* Entry point outside code segment
```

The entry point of the module is stored as an offset in the code segment, and therefore should be within a reachable part of it. In this module it does not. This error probably implies, that the entry point is patched manually.

```
MODRUN - *ERROR* premature end of file at <fileptr>
```

The file is corrupt, or not a loadmodule. < fileptr > is a decimal number, denoting the pointer in the file, where MODRUN attempted to read.

```
MODRUN-*ERROR* <name> is not a load module
```

MODRUN discovered that the file is corrupt or not a load module before it attempted to read outside the file.

```
MODRUN-*ERROR* Wrong Code size (355c) or BlockTag (47)
```

This means, that the file is a loadmodule, but it is corrupted one way or another. A wrong BlockTag is read, which can mean, that the code size is wrong, because BlockTag should immediately follow the code segment. BlockTag should be 21 (hex).

```
MODRUN-*ERROR* DataSize (340) smaller than initialised
data area (4b9c7)
```

Just another way to tell that the module is corrupted.

Runtime Libraries

General remarks

Introduction

This chapter of the Par.C System Manual contains a reference of the Par.C runtime facilities.

The section on header files gives a description of the contents of the header files and explanations of the types and macros defined in these files. Although some of the types and macros are internal to the Par.C System, they are included in the explanatory lists.

The Library reference contains descriptions of all standard Par.C functions, ordered alphabetically. An underscore character or a sequence of dots in the function name is ignored.

Although we have tried to adjust documentation and implementation of the Par.C System runtime libraries to each other, there is always the possibility of errors, omissions or differences. If you find errors of this kind in this manual, please notify us via the bug report form included in the appendices.

Compatibility with Draft proposed ANSI C

The Par.C runtime libraries contain most of the Draft Proposed ANSI C facilities, although some of these cause problems in relation to parallel processing. The deviations from Draft Proposed ANSI C are listed below.

- `locale.h` is not available. The use of different locales is not supported in the current version of Par.C.
- `setjmp.h` is not available. The use of the functions contained herein causes serious problems in concurrent processing. It is not yet clear if these functions can be implemented at all in the Par.C System.
- Support for temporary files is not available. This implies that `tmpfile()` and `tmpnam()` cannot be used, `L_TMPNAM` is not specified and `TMP_MAX` is not declared.
- Non-defined macros are: `EDOM`, `ERANGE`, `HUGE_VAL`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, and `SIGTERM`.
- The types `sig_atomic_t`, `fpos_t`, `div_t` and `ldiv_t` are not defined.

- According to Draft Proposed ANSI C, the return types and types of arguments of some facilities should be `void *`. In the current version of the Par.C System, the compiler will not allow operations on `void *` types. Therefore these types have been defined as `char *`.

Apart from the missing functions named and implied above, the non-available functions in the current version of the Par.C System are the following:

- `freopen()`, `fgetpos()`, `fsetpos()`, `div()`, `ldiv()`, and `strcoll()`.

These functions have not been implemented yet, but will probably be available in the next version of the Par.C System.

- `strftime()`

This function is related to the support of different locales via locale.h and is therefore not implemented in the current version of the Par.C System.

The following functions are implemented conform to Draft Proposed ANSI C, but have been given a re-entrant replacement which fits better into a parallel processing environment.

ANSI C function	**Par.C System replacement**
`getenv()`	`getenv_r()`
`gmtime()`	`gmtime_r()`
`asctime()`	`asctime_r()`
`localtime()`	`localtime_r()`
`strtok()`	`strtok_r()`
`strerror()`	`strerror_r()`

Header files

assert.h

Contents: .... `assert()`

Description: ..... `assert.h` is used in debugging C programs and contains the definition of `assert(p)`. When the value of `p` is false or zero, this macro can be used to give a diagnostic message and exit. The message will indicate where in your program the error occurred. If the macro `NDEBUG` is defined when `assert.h` is included, `assert(p)` is defined as the empty string, thereby annulling the effect of `assert(p)` in the program.

crash.h

This file has been removed from the Par.C System. The routine `CRASH()` is replaced by a more sophisticated fatal error handling utility, for internal use by the Par.C System only.

ctype.h

Contents: ....

`isalnum()`	`isgraph()`	`isspace()`	`tolower()`
`isalpha()`	`islower()`	`isupper()`	`toint()`
`isascii()`	`isodigit()`	`iswhite()`	`toupper()`
`iscntrl()`	`isprint()`	`isxdigit()`	`_tolower()`
`isdigit()`	`ispunct()`	`toascii()`	`_toupper()`

Description: ..... `ctype.h` contains declarations of facilities for character processing. These facilities are of two kinds: classification and conversion.

The classification facilities check on the inclusion of the character in a specific class of characters, returning a nonzero integer if the character falls inside the class to be considered and zero otherwise. All classification facilities declared in `ctype.h` are implemented as functions, and are available in `std.lib`.

The conversion facilities cause the character to fall inside a specific range, by changing its value where needed. All conversion facilities declared in `ctype.h` return an integer value representing a character or EOF. All conversion facilities are implemented as functions, and are available in `std.lib`, except for `toascii()` which is implemented as a macro.

Notes and remarks: `_tolower` and `_toupper` are versions of `tolower()` and `toupper()`, which are implemented as macros and allow only a restricted range of characters as arguments. Since these macros do not do any checking, they could produce unwanted results if the argument is not a letter.

errno.h

Contents: .... Error codes used in the Par.C System.

Description: ..... `errno.h` contains definitions of errorcodes. Most errorcodes concern errors that may occur while doing file I/O. Other standard errorcodes are defined for memory manager errors and errors in math functions.

A full list with errorcode values and explanations is given below:

Fatal system errors:

ESYS_MEMCORR	Memory structure corrupted
ESYS_OUTOFMEM	Out of memory
ESYS_CORRUPT	Process structure corrupted
ESYS_ILLCALL	Illegal system call
ESYS_INTERN	General internal error (Bug)
ESYS_COMFAIL	Fatal communication failure
ESYS_DEADLOCK	Unrecoverable deadlock

Recoverable errors:

E_EOF	End of file
E_UNKACC	Unknown access code
E_NOPNFWR	Not open for Write
E_NOPNFRD	Not open for Read
E_NOSUFI	No Such File
E_READ	Read error
E_WRITE	Write error

E_CANTSEND	Cannot send over channel/link
E_CANTRECV	Cannot receive from link/channel
E_DISKFULL	Disk full
E_WRITPROT	Medium is write protected
E_CANTOPEN	Cannot open file
E_CANTSEEK	Cannot seek
E_NOTCON	FILE structure not connected to open file
E_UNKREQ	Unknown request code received
E_TIMEOUT	Time_out on communication
E_ILPFDB	illegal pointer to FDB
E_IGNORED	Command ignored, probably destination inactive
E_ILLFID	Illegal File ID
E_ABORTED	Command aborted
E_DRINUSE	Driver still in use
E_UNKDEV	Unknown device (not installed)
E_CANTALLOC	Memory allocation failure
E_CANTFREE	Memory deallocation failure
E_CANTCLOSE	File can not be closed properly
E_CANTTELL	File pointer not available
E_LINKNO	Attempt to access non-existing link (LinkNo<0 or LinkNo>3)
E_CANTREMOVE	Failed to delete file
E_CANTRENAME	Failed to remove file
E_ILLPARM	Illegal parameter value
E_CANTWRITE	Currently reading, can not write without seek
E_CANTREAD	Currently writing, can not read without seek
E_CANTINSTALL	Failed to install driver
E_CANTUNGET	unget without get or double unget
E_RSLFMT8	realt8.rsl : format corrupted
E_NOREALT8	Can not find resident library realt8.rsl
E_RSLFMT4	realt4.rsl : format corrupted
E_NOREALT4	Can not find resident library realt4.rsl
E_UNDEFREF	Call to undefined reference
E_COMFAIL	Communication failure
E_GIOERR	General I/O error

Math errors:

EM_GENERAL	General math error
EM_LOGDOMAIN	Argument of logarithm <= 0

`EM_POW2UFLOW`	Result of `pow2` too small
`EM_POW2OFLOW`	Result of `pow2` too large
`EM_COSHOFLOW`	Result of `cosh` too large
`EM_SINHOFLOW`	Result of `sinh` too large
`EM_SQRTDOMAIN`	Argument of `sqrt < 0`
`EM_LDEXPOFLOW`	`ldexp` overflow
`EM_LDEXPUFLOW`	`ldexp` underflow
`EM_POWDOMAIN`	`pow` : illegal arguments
`EM_POWOFLOW`	Result of power too large
`EM_LOG10DOMAIN`	Argument of `log10 <= 0`
`EM_ATAN2UNDEF`	`atan2` undefined
`EM_ASINDOMAIN`	`asin` not defined outside [-1,1]
`EM_ACOSDOMAIN`	`acos` not defined outside [-1,1]
`EM_ACOSHDOMAIN`	`acosh` not defined below -1
`EM_ATANHDOMAIN`	`atanh` not defined outside [-1,1]
`EM_POW10UFLOW`	Result of `pow10` too small
`EM_POW10OFLOW`	Result of `pow10` too large
`EM_EXPUFLOW`	Result of `exp` too small
`EM_EXPOFLOW`	Result of `exp` too large
`EM_MODFIOFLOW`	Integer overflow in `modf`
`EM_POWUFLOW`	Result of `pow` too small

float.h

Contents: .... Characteristics of the `float` and `double` data types.

Description: ..... `float.h` contains the implementation-dependent characteristics of the `float` and `double` datatypes. The format of these datatypes determines their precision. In the Par.C System, full transparency is maintained regarding floating point arithmetic on T4xx and T8xx transputers. This implies that the format of float and double variables is conform the format used in the T8xx's FPU, which largely conform to the IEEE 754 format definitions [Inmos 1986, IEEE 1985].

A list of definitions with explanations is given below.

`FLT_MAX`	max. representable finite nr.	3.402823466e+38
`FLT_MIN`	min. normalised positive nr.	1.175494351e–38
`FLT_EPSILON`	smallest significant value	1.192092896e-07
`FLT_MAX_10_EXP`	maximal decimal exponent	38
`FLT_MAX_EXP`	maximal binary exponent	128

`FLT_MIN_10_EXP`	miminal decimal exponent	-37
`FLT_MIN_EXP`	minimal binary exponent	-125
`FLT_MANT_DIG`	number of bits in mantissa	24
`FLT_DIG`	precision (decimal digits)	6
`FLT_ROUNDS`	addition rounding	0
`FLT_RADIX`	exponent radix	2
`DBL_MAX`	max. repr. finite number	1.7976931348623158e+308
`DBL_MIN`	min. normalised pos. number	2.2250738585072014e-308
`DBL_EPSILON`	smallest significant value	2.2204460492503131e-016
`DBL_MAX_10_EXP`	maximal decimal exponent	308
`DBL_MAX_EXP`	maximal binary exponent	1024
`DBL_MIN_10_EXP`	minimal decimal exponent	-307
`DBL_MIN_EXP`	minimal binary exponent	-1021
`DBL_MANT_DIG`	number of bits in mantissa	53
`DBL_DIG`	precision (decimal digits)	15
`DBL_ROUNDS`	addition rounding	0
`DBL_RADIX`	exponent radix	2

The smallest significant value given is defined as the smallest value `x` for which `1.0 + x != 1.0`. Note that there is an important difference with the smallest representable value.

Notes and remarks: `float` arithmetic uses the `float` format as long as no functions are called. Whenever the `float` variable is listed as parameter for a function, a type conversion to `double` is performed.

The codes used by the T8xx for Not-a-Numbers are also supported in the primitive floating point arithmetic operations implemented for the T4xx and included in the realt4.rsl resident library. A list of meanings of the NaN-codes is given below.

Single length	**Double length value**	**Error signified**
7FC00000	7FF80000 00000000	Divide zero by zero
7FA00000	7FF40000 00000000	Divide infinity by infinity
7F900000	7FF20000 00000000	Multiply zero by infinity
7F880000	7FF10000 00000000	Addition of opposite signed infinities or subtraction of same signed infinities
7F840000	7FF08000 00000000	Negative square root
7F820000	7FF04000 00000000	64 to 32 bit NaN conversion
7F804000	7FF00800 00000000	Remainder from infinity
7F802000	7FF00400 00000000	Remainder by zero

The NaN's listed above are generated by the T8xx's hardware and by the FPU-emulating functions running on the T4xx. The library functions will, when an error occurs, return the following NaN with an error code in the low word.

Example: .... 7FF00000 00001015 NaN for `sqrt()` domain error.

limits.h

Contents: .... Miminum values and extra definitions of the standard types in the Par.C System.

Description: ..... A list of definitions of the standard types in the Par.C System is given below.

`CHAR_BIT`	width in bits of char type	8
`SCHAR_MIN`	minimum value of signed char	-128
`SCHAR_MAX`	maximum value of signed char	127
`UCHAR_MAX`	maximum value of unsigned char	255
`SHRT_MIN`	minimum value of short int	-32768
`SHRT_MAX`	maximum value of short int	32767
`USHRT_MAX`	maximum value of unsigned short	65535
`INT_MIN`	minimum value of int	-2147483648
`INT_MAX`	maximum value of int	2147483647
`UINT_MAX`	maximum value of unsigned int	4294967295
`LONG_MIN`	minimum value of long int	-9223372036854775808
`LONG_MAX`	max. value of long int	9223372036854775807
`ULONG_MAX`	max. value of unsigned long	18446744073709551615
`CHAR_MIN`	is defined as `SCHAR_MIN`	
`CHAR_MAX`	is defined as `UCHAR_MAX`	

math.h

Contents: ....

`acos()`	`exp()`	`log10()`	`sqrt()`
`asin()`	`fabs()`	`modf()`	`tan()`
`atan()`	`floor()`	`pow()`	`tanh()`
`atan2()`	`fmod()`	`pow2()`	`matherr()`
`ceil()`	`frexp()`	`pow10()`	`_matherr()`
`cos()`	`ldexp()`	`sin()`	`NaN()`
`cosh()`	`log()`	`sinh()`	

Description: ..... `math.h` contains declarations of floating point arithmetic functions. All of these functions are available in `math.lib` and make use of the primitive floating point arithmetic operations included in the resident libraries `realt4.rsl` and `realt4.rsl`. This implies that all functions are fully transparent regarding the type of transputer used.

All functions use the `double` datatype and guarantee a maximum precision obtainable using that type. The return value is in the double format. Errors will cause a Not-a-Number (`NaN`) to be returned. A list of possible `NaN`'s and their meanings is given in the description of `float.h`. You can check if a double is a `NaN` by using the function `NaN()`.

servasm.h

Because the Par.C file I/O system has been changed and will be extended further in the next release of the Par.C System, the `servasm.h` and `server.h` include files have been removed. New facilities and documentation for writing device drivers and custom servers will be provided with the next release.

server.h

Because the Par.C file I/O system has been changed and will be extended further in the next release of the Par.C System, the `servasm.h` and `server.h` include files have been removed. New facilities and documentation for writing device drivers and custom servers will be provided with the next release.

signal.h

Contents: ....

`ABORT_SIG`	`SIG_DFL()`	`MAX_SIG`	`HIGH_PRIORITY`
`EVENT_SIG`	`SIG_IGN()`	`raise()`	`LOW_PRIORITY`
`ALARM_SIG`	`SIG_ERR()`	`signal()`	

Description: ..... `signal.h` contains declarations of functions related to the signal function in C, which provides a simple mechanism for exception handling. This mechanism allows programmers to implement a signal handling function, which is activated when the signal related to it is triggered.

`signal()` is used to relate the signal handling function to a specified signal value. The various signals can be triggered by the hardware (via the computer's error-detection mechanism) or by the software (calling `raise()` with the specified signal value as argument). The latter method enables a process to activate a signal handling process, without needing any software linkage between the two except for the signalname.

`SIG_DFL()`, `SIG_IGN()` and `SIG_ERR()` are defined as functions to correspond to the arguments expected by `signal(sig, (*func)())`. For a description of their use, see the description of `signal()`.

stdarg.h

Contents: ....

`va_list`	`va_start()`	`va_end()`
`_va_arg()`	`va_arg()`	

Description: ..... The facilities declared in `stdarg.h` provide a standard way of accessing variable argument lists. These are needed to support certain functions (such as `fprintf()` and `vfprintf()` and possible user-defined functions).

`va_list` is a type for local state variables, which are used in traversing the argument list. The local state variable is usually called `ap`.

`va_start()`, `va_arg()` and `va_end()` are defined as macros (according to the Draft Proposed ANSI C standard), and are described in the Library reference section.

stdcnv.h

Contents: ....

`itoa()`	`ultoa()`	`atou()`	`strtol()`
`utoa()`	`xltoa()`	`atox()`	`strtoul()`
`xtoa()`	`Xltoa()`	`atoo()`	`_toa()`
`Xtoa()`	`oltoa()`	`atob()`	`_ato()`
`otoa()`	`bltoa()`	`atof()`	`_flt`
`btoa()`	`dtoa()`	`atol()`	
`ltoa()`	`atoi()`	`atoul()`	

Description: ..... `stdcnv.h` contains a large number of conversion facilities. Most of these have been implemented as macros, expanding to a call to the generic conversion functions `_toa()` and `_ato()`. These functions require a pointer to the string concerned, a base value with possible flags for signed or unsigned types, and a value or variable.

`_flt` is a structure type defined as an "unpacked" floating point number, containing integers for sign and exponent and an unsigned long for the mantissa.

Notes and remarks: The `atoi()`, `atol()` and `atof()`, `strtol()` and `strtoul()` facilities (all implemented as macros) are also defined in `stdlib.h`, in accordance with the Draft Proposed ANSI C standard. Since conditional compiling is used, including both `stdlib.h` and `stdcnv.h` will not lead to warnings or error messages. The function `strtod()` is declared in both header files as well, and will not lead to warnings or error messages either.

stddef.h

Contents: ....

`size_t`	`_Byte`	`MostPos`
`ptrdiff_t`	`_Word`	`V()`
`errno`	`BytesInWord`	`MinInt`
`offsetof()`	`P()`	`_Tn`
`NULL`	`__PROG__`	`_nT`
`semaphore`	`__CSEG__`	`_ttype`
`TRUE`	`__DSEG__`	`LinksPerTransputer`
`FALSE`	`MostNeg`	

Description: ..... `stddef.h` contains declarations of standard definitions, and should be included by programs using any of these. Included are general purpose definitions, as well as definitions and declarations specific for the Par.C System.

Notes and remarks: This file is included in the following files:

`assert.h`	`stdarg.h`	`stdlib.h`	`transp.h`
`errno.h`	`stdcnv.h`	`system.h`	
`signal.h`	`stdio.h`	`time.h`	

Therefore, if any of these files is included, `stddef.h` is automatically included as well.

Explanation of definitions and declarations

size_t

ANSI C type, denoting the unsigned type of the result of the `sizeof` operator. In the Par.C System, `size_t` is defined as `unsigned int`.

ptrdiff_t

ANSI C type, denoting the type of the result of subtracting two pointers. In the Par.C System, `ptrdiff_t` is defined as `int`.

errno

This variable should contain the code assiociated with the last runtime error encountered. Although this may be useful for sequential processes, it is unreliable when the concurrent features of the Par.C System are used. A more sophisticated error handling mechanims designed for concurrent processing, is still under development.

offsetof(s, f)

ANSI C definition, evaluating to the offset of the field denoted by `f` inside the structure denoted by `s`. `offsetof()` is implemented as a macro.

NULL

The `NULL` pointer. To keep programs upward compatible with future versions of the Par.C System, it is highly recommended to use this macro instead of the value 0, because in future versions `MostNeg` or `MostPos` might be used as `NULL` pointer value. This is the result of the transputer addressing its memory in a signed space starting at `MostNeg`. This implies that 0 can represent a valid pointer.

semaphore

Type definition for semaphore variables. A description of the use of semaphores is given in the library reference to the `P()` and `V()` functions.

TRUE

An integer with value 1 used as boolean.

FALSE

An integer with value 0 used as boolean.

_Byte

Defined as `unsigned char`. The default character type of the Par.C compiler is signed. However, many operations and type conversions are more efficient using unsigned characters.

_Word

Defined as `unsigned int`. The default integer type of the Par.C compiler is signed. However, many values require unsigned integers.

BytesInWord

Par.C System definition. This macro evaluates to the number of bytes in a transputer word. In the current version of the Par.C System this value is fixed at 4 (the length in bytes of a 32-bit word).

__PROG__

Global variable containing a pointer to the name of the program. This is the same pointer as `argv[0]` (second parameter of main).

__CSEG__

Global variable containing a pointer to the start of the code segment, and can be used to calculate a relative offset inside the code segment.

__DSEG__

Global variable containing a pointer to the start of the data segment, and can be used to calculate a relative offset inside the data segment.

MostNeg

The largest negative number representable in a signed transputer word. In the current version of the Par.C System this value is fixed at 0x80000000 (the most negative 32-bit integer).

MostPos

The largest positive number representable in a signed transputer word. In the current version of the Par.C System this value is fixed at 0x7FFFFFFF (the most positive 32-bit integer).

MinInt

Stands for minimum integer and has the same value as `MostNeg`. `MinInt` is often used in transputer reference manuals and Inmos technical notes.

_Tn

Par.C variable. When the program is loaded into each transputer in the network, this variable is initialised to the number of the current transputer as provided by the network loader. Note that the `_Tn` variable will have no effect in programs linked with `a.lib`, since these programs will only be loaded on one single transputer, and `_Tn` will therefore always have the value 1.

_nT

Par.C variable. When the program is loaded into each transputer in the network, this variable is initialised to the total number of transputers in the network, as found by the network loader. When the program is linked with `a.lib`, `_nT` will have the value 1.

_ttype

Par.C variable. When the program is loaded into each transputer in the network, this variable is initialised to indicate the type of the current transputer. The value of `_ttype` is 2 on a T212, 4 on a T414, 5 on a T425 and 8 on a T800.

LinksPerTransputer

Par.C definition. In the current version of the Par.C System this value is fixed at 4.

Notes and Remarks The use of the Par.C variables `_Tn`, `_nT` and `_ttype` is discouraged. For future compatability it is advised to use the equivalent fields in the `SYSTEM` structure as defined in `system.h`.

stdio.h

Contents: ....

_IOFBF	DIR	fputc()	puts()
_IOLBF	stderr	fputs()	putc()
_IONBF	stdin	fread()	putchar()
BUFSIZ	stdout	fscanf()	remove()
LBUFSIZ	clearerr()	fseek()	rename()
EOF	Dir()	ftell()	rewind()
FILE	DirInit()	fwrite()	Run()
FNULL	fclose()	getc()	scanf()
_F_ERROR	ferror()	getchar()	setbuf()
_F_EOF	feof()	gets()	setvbuf()
OK	fflush()	InPort()	sprintf()
SEEK_CUR	fgetc()	OutPort()	sscanf()
SEEK_END	fgets()	PeekHost()	ungetc()
SEEK_SET	filerr()	perror()	vfprintf()
MAX_LINE	fopen()	PokeHost()	vprintf()
MAX_OPEN	fprintf()	printf()	vsprintf()

Description: ..... `stdio.h` contains the necessary definitions and declarations to enable the use of standard input/output facilities. All names in capitals are macro definitions and are explained below. Functions are explained in the Library reference chapter, where all runtime library functions are listed in alphabetical order.

The functions in stdio.h provide a standard way of communicating with the "outside world". In the present version of the Par.C System, these file I/O functions can only be called directly on the first transputer in the network, since only this first transputer has a host driver process running to support file I/O. On other transputers calling these functions will probably result in a crash.

Notes and remarks: The Par.C file I/O system is capable of handling multi-process file I/O. Therefore some slight changes and additions have been made to the standard facilties included in `stdio.h`.

Explanation of definitions and declarations

_IOFBF

Calling `setvbuf()` with `_IOFBF` causes the buffering mode for the specified stream to be set to full buffering. This implies that the buffer is completely filled or emptied before writing or reading characters via the file I/O system. This type of buffering is useful when acessing disk files.

_IOLBF

Calling `setvbuf()` with `_IOLBF` causes the buffering mode for the specified stream to be set to line buffering. This implies that characters are written or read whenever a newline or an `EOF` is encountered, or whenever the buffer is completely filled or emptied. This type of buffering is useful when acessing the screen.

_IONBF

Calling `setvbuf()` with `_IONBF` causes the buffering mode for the specified stream to be set to no buffering. This implies that characters are written and read directly via the file I/O system. This type of buffering is useful whenever direct transfer is wanted.

BUFSIZ

`BUFSIZ` gives the default size of the buffer used by the `setbuf()` function (full buffering).

LBUFSIZ

`LBUFSIZ` gives the default size of the buffer when using the line buffering mode.

EOF

EOF expands to a negative value that is returned by several functions to indicate an end_of_file being reached.

FILE

FILE denotes a structure containing all necessary information on a file which can be accessed by calling the appropriate file I/O functions. The structure contains a number of flags indicating the opening mode of the file, a pointer to the last character read from or written to the file, a variable holding information on a possible error encountered while reading from or writing to the file, a file pointer, a buffer pointer and some information needed by the file I/O system.

FNULL

FNULL is used as pointer to a non-existant file. Its value is equal to the NULL pointer, but cast to a FILE pointer.

_F_ERROR

_F_ERROR is used to indicate an error while reading from or writing to a file, except for an EOF being reached.

_F_EOF

_F_EOF is used to indicate an EOF being reached while reading.

OK

OK is used to indicate that no errors have been encountered while reading from or writing to a file.

SEEK_CUR

SEEK_CUR is used to indicate a reference point for the `fseek()` function. When calling `fseek()` with SEEK_CUR as third argument, the position is set to a position with the indicated offset from the current position of the file pointer.

SEEK_END

SEEK_END is used to indicate a reference point for the `fseek()` function. When calling `fseek()` with SEEK_END as third argument, the position is set to a position with the indicated offset from the end of the file.

SEEK_SET

SEEK_SET is used to indicate a reference point for the fseek() function. When calling fseek() with SEEK_SET as third argument, the position is set to a position with the indicated offset from the start of the file.

MAX_LINE

MAX_LINE is an integer which is set to indicate the maximum length of a string, used in the file I/O system. This variable can be used to avoid overflow when calling gets(). It may be altered to influence gets().

MAX_OPEN

MAX_OPEN indicates the maximum number of files which can be open simultaneously. In the current version of the Par.C System, this number is set to MostPos (the largest representable integer), since the structure of the file I/O system does not impose any restrictions on the number of opened files. However, the host system you are working on might set such restrictions.

DIR

DIR is defined as a structure containing the directory information which is received from the host through the use of Dir().

stderr

This variable contains a pointer to the file descriptor of the standard error stream.

stdin

This variable contains a pointer to the file descriptor of the standard input stream.

stdout

This variable contains a pointer to the file descriptor of the standard output stream.

stdlib.h

Contents: ....

RAND_MAX	_exit()	strtod()
MAX_ONEXIT	free()	strtol()
MEMINFO	getenv()	strtoul()
abort()	getenv_r()	calloc()
_abort()	labs()	realloc()
abs()	malloc()	system()
atexit()	onexit_t	RunProcess()
atof()	onexit()	Release()
atoi()	qsort()	SendLink()
atol()	rand()	RecvLink()
bsearch()	realloc()	SendLinkOrFail()
calloc()	smalloc()	RecvLinkOrFail()
exit()	srand()	MemAvail()

Description: ..... `stdlib.h` contains declarations of standard facilities, which can be divided into three different kinds:

The first group of facilities is used in storage allocation, and consists of the functions `calloc()`, `malloc()`, `smalloc()`, `realloc()` and `free()`.

The second group of facilities gives control over the program's execution and termination in different ways, and consists of the functions `abort()`, `atexit()`, `exit()`, `_exit()`, `onexit()`, the type `onexit_t` and the macro `MAX_ONEXIT`.

The remainder of `stdlib.h` contains declarations of various elementary functions, including some simple arithmetic, some much-used conversion utilities and the `getenv()` and `getenv_r()` routines. The macro `RAND_MAX` is used in the `rand()` and `srand()` functions.

`struct MEMINFO` is defined as a structure keeping information on the available free memory blocks, as returned by `MemAvail()`.

string.h

Contents: ....

`memchr()`	`strcpy()`	`strpcbrk()`
`memcmp()`	`strcspn()`	`strrchr()`
`memcpy()`	`strerror()`	`strrpbrk()`
`memfill()`	`strerror_r()`	`strrpos()`
`memmove()`	`strlen()`	`strspn()`
`memset()`	`strncat()`	`strstr()`
`strcat()`	`strncmp()`	`strtok()`
`strchr()`	`strncpy()`	`strtok_r()`
`strcmp()`	`strpbrk()`	`MAX_ERRMSG_LENGTH`

Description: ..... `string.h` contains declarations of string handling facilities. The functions with names starting with "`mem`" act on a fixed specified number of characters, starting from the pointers which are given as arguments when calling the functions. The functions with names starting with "`str`" act on strings and check on a terminating null character.

The function `strtok_r()` is a re-entrant version of `strtok()`, which is by definition not re-entrant. When more than one process on the same processor uses `strtok()`, the result is not defined. The re-entrant version guarantees correct behaviour in this case. The functions `strerror_r()` and `strerror()` are related in the same way. See the descriptions of these functions for more details.

system.h

Contents: ....

`SYSTEM`	`_SetParStack()`	`GetSysInfo()`
`NODEDESCRIPTOR`	`LinkStatus`	`Priority()`
`_GetFunStack()`	`HOST_CONNECTED`	`SetPriority()`
`_SetFunStack()`	`NOT_CONNECTED`	`_tohost`
`_GetParStack()`	`GetNodeInfo()`	`_fromhost`

Description: ..... `system.h` contains transputer-dependent facilities of the Par.C System. For a description of the functions mentioned above see the Library

Reference. The types and macros defined in `system.h` are explained below.

SYSTEM

This type describes a structure containing information on the transputer network on which the program is running, and on the specific characteristics of the current node. A more detailed description of the `SYSTEM` structure can be found in the Tutorial.

NODEDESCRIPTOR

This type describes a structure containing the local network information in a packed form.

LinkStatus

The network analyser in the Par.C boot system uses a number of `LinkStatus` codes, which are defined in `system.h` and will not be listed here.

HOST_CONNECTED

This is a constant value used in the `Network[]` array in the `SYSTEM` structure, used to indicate the link connected to the host system. This value is therefore only used on the root transputer (#1).

NOT_CONNECTED

This is a constant value used in the `Network[]` array in the `SYSTEM` structure, used to indicate an unconnected link.

_tohost

This is a channelpointer containing the address of the outgoing bootlink of the current transputer (which indicates the direction of the host).

_fromhost

This is a channelpointer containing the address of the incoming bootlink of the current transputer (which indicates the direction from the host).

time.h

Contents: ....

`AFTER()`	`_alarm()`	`localtime()`
`BEFORE()`	`asctime()`	`localtime_r()`
`CLK_TCK`	`asctime_r()`	`mktime()`
`_FIRSTDAY`	`clock()`	`_sleep()`
`_BASEYEAR`	`ctime()`	`sleep()`
`clock_t`	`DateToInt()`	`time()`
`_daylight`	`delay()`	`wait()`
`time_t`	`difftime()`	`yday()`
`_timezone`	`gmtime()`	`delay()`
`tm`	`gmtime_r()`	`_TicksPerSecond()`
`alarm()`	`IntToDate()`	

Description: ..... `time.h` contains declarations of facilities used in time and date handling. The macros and types supporting these facilities are explained below.

A number of time and date facilities will give rise to problems when they are used in the Draft Proposed ANSI C standard form in concurrent processes, since they are non re-entrant. These funtions have been given re-entrant replacements, which can be easily recognized by the "_r" suffix in the name of the function, i.e. the `localtime_r()` function is the re-entrant replacement of the Draft Proposed ANSI C standard function `localtime()`.

The declared macros and types are explained in the following list:

CLOCKS_PER_SEC

This macro expands to the number of processor clockticks per second, and can be used to calculate periods of time in seconds rather than in processor clock ticks. Since the transputer contains one processor clock for each level of priority, the value of `CLOCKS_PER_SEC` cannot be determined at compiletime, and is expanded to a function call, returning a priority-dependent value of `CLOCKS_PER_SEC`.

_FIRSTDAY

This macro is used in some of the time and date functions to indicate the starting date for date measurement and calculation. In the current version of the Par.C System, the value of `_FIRSTDAY` is set to 3.

_BASEYEAR

This macro is used in some of the time and date functions to indicate the starting year for date measurement and calculation. In the current version of the Par.C System, the value of `_BASEYEAR` is set to 1972.

clock_t

This type indicates the type of a timer value, returned by some of the time-related functions, such as `clock()`. In the current version of the Par.C System, `clock_t` is defined as an integer.

_daylight

This variable is used in combination with time functions to indicate whether daylight savings time is in force or not. If this variable is needed, the environment variable `SUMMERTIME` is read when the program is loaded. If `SUMMERTIME` is defined, one hour is added to the calculated localtime, regardless of the value given to the variable on the host system.

time_t

This type is used by some of the time-related functions, such as `mktime()`. In the current version of the Par.C System, `time_t` is defined as an integer.

_timezone

This variable is used in combination with time functions to indicate which timezone is to be used. If this variable is needed, the environment variable `TIMEZONE` is read to obtain the correct value of `_timezone` when the program is loaded. The correct format for `TIMEZONE` is: `[-]H[:M]`, where `0 =< H < 12` and `0 =< M < 60`. This value should indicate the difference between the local time and Greenwich Mean Time.

struct tm

This type is defined as a structure, holding all necessary information on the calendar time. The structure contains integers for the number of seconds after the minute, the number of minutes after the hour, the number of hours since midnight, the day of the month, the number of months since January, the year, the number of days since Sunday, the number of days since December 31 of the previous year, and an indication for daylight savings time. The structure is used by functions related to date manipulation.

transp.h

Contents: ....

`LOW_PRIORITY`	`Deschedule()`	`_TestClrErr()`
`HIGH_PRIORITY`	`_SetHaltErr()`	`_STLF()`
`LINKOUT(X)`	`_ClrHaltErr()`	`_STLB()`
`LINKIN(X)`	`_ResetSystemTimers()`	`_STHF()`
`EVENT`	`_StopProcess()`	`_STHB()`
`ProcessDescriptor`	`ResetChannel()`	`_LDLF()`
`NotProcessP`	`_RestartProcess()`	`_LDLB()`
`_in()`	`_SetErr()`	`_LDHF()`
`_out()`	`_TestErr()`	`_LDHB()`

Description: ..... The routines declared in `transp.h` are all low-level and specific to the transputer hardware. Most of these will be obvious in their use. The routines names in the rightmost column should only be used with sufficient knowledge of the transputer's hardware, since their usage may interfere with the standard scheduling mechanism of the transputer.

More information on the use of the routines listed above is given in the Library Reference. It is possible that some of the declared functions are not described in this edition of the manual.

Par.C Library Reference

abort()

Usage: ....
```
#include <stdlib.h>
void abort();
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `abort()` causes abnormal program termination, unless the signal `SIGABRT` is caught by some signal handling routine which does not return. The `abort()` function cannot return to its caller.

Return values:.... none

See also: ....
`_abort()`
`exit()`
`atexit()`

_abort()

Usage: ....

```
#include <stdlib.h>
void _abort();
```

Availability: ..... Par.C System

Libraries: .... `io.lib` and `noio.lib`

Description: ..... `_abort()` causes abnormal program termination by sending an abort command to the server on the transputer (named hostdriver). The hostdriver in turn sends an abort command to the server on the host (named `run`), which will exit with exit value -2. In contrast to `abort()`, `SIGABRT` is not raised. The `_abort()` function cannot return to its caller. run will display the message :

```
<progname> - *FATAL* : Program aborted with code -2
```

Return values: ... none

Notes and remarks: The only difference between `abort()` and `_abort()` is that `SIGABRT` is not raised.

This routine is extremely crude: no files are closed and no graceful termination is performed.

See also: ....

```
abort()
exit()
atexit()
```

abs()

Usage: ....
```
#include <stdlib.h>
int abs(x);
int x;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `abs()` computes the absolute value of `x`, which is expected to be a (signed) integer.

Return values:.... `abs()` returns the absolute value of `x`.

See also: ....
`labs()`
`fabs()`

acos()

Usage: ....
```
#include <math.h>
double acos(x);
double x;
```

Availability: ..... Draft Proposed ANSI C

Libraries: .... `math.lib` and `t8math.lib`

Description: ..... `acos()` computes the arc cosine of `x`. The argument `x` is expected to have the type double.

Return value: .... `acos()` returns the value of the arc cosine of `x`.

AFTER()

Usage: ....
```
#include <time.h>
int AFTER(T1, T2)
time_t T1, T2;
```

Availability: ...... Par.C System

Description: ..... The macro `AFTER()` compares two times and reports if `T2` precedes `T1`.

Return values:.... `AFTER()` returns zero if `T2` does not precede `T1`, otherwise a non-zero value is returned.

See also: .... `BEFORE()`

alarm()

Usage: ....

```
#include <time.h>
int alarm(n);
int n;
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: ..... `alarm()` sets an internal system timer to the indicated number of seconds and returns the number of seconds previously on this timer. When the timer expires, the signal `SIGALRM` is raised, causing the signal handling routine which is associated with this signal to be executed. If `n` is zero the effect of the call is to "reset" the alarm, i.e. to cancel previous calls to `alarm()`.

Notes and remarks: `alarm()` will cause all processes which are at that moment inactive due to a call to `sleep()` or `_sleep()` to resume execution immediately, whereby the interrupted `sleep()` returns the number of unslept seconds or clockticks, respectively. Raising `SIGALRM` by calling `raise()` will *not* cause 'sleeping' processes to be woken up.

Return values: ... `alarm()` returns the number of seconds which was on the timer before the call to `alarm()` was made.

See also: ....

```
_alarm()
signal()
raise()
sleep()
_sleep()
```

_alarm()

Usage: ....
```
#include <time.h>
int _alarm(n);
int n;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `_alarm()` sets an internal system timer to the indicated number of clockticks and returns the number of clockticks previously on this timer. If `n` is zero the effect of the call is to "reset" the alarm, i.e. to cancel previous calls to `_alarm()`.

Notes and remarks: `_alarm()` differs from `alarm()` in that the value of `n` is taken as low-priority clockticks instead of seconds.

`_alarm()` will cause all processes which are inactive due to a call to `sleep()` or `_sleep()`, to resume execution after `n` ticks. Raising `SIGALRM` by calling `raise()` will *not* cause 'sleeping' processes to be woken up.

Return values:.... `_alarm()` returns the number of clockticks which was on the timer before the call to `_alarm()` was made.

See also: ....
```
alarm()
signal()
raise()
sleep()
_sleep()
```

asctime()

Usage: ....

```
#include <time.h>
char *asctime(ptr);
struct tm *ptr;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `asctime()` converts the broken-down time given in the `tm` structure pointed to by `ptr` to an ASCII string in the following format:

DAY MON day hour:min:sec YEAR

A newline and the terminating null character are appended to the string formed. An example of the format: Mon Dec 19 18:15:00 1988

Return values: ... `asctime()` returns a pointer to an ASCII string denoting the calendar time given in the `tm` structure pointed to by `ptr`.

Notes and remarks: Since this and related time and date functions have been defined to operate on a structure and a character array in the static data area, calls to other functions may overwrite the result of a call to `asctime()`. Also, other processes calling the same routine will cause the result to be overwritten. To avoid these effects all non re-entrant time and date functions have been given a re-entrant replacement in the Par.C System. The replacement for `asctime()` is `asctime_r()`.

See also: ....

```
asctime_r()
time()
time.h
```

asctime_r()

Usage: ....

```
#include <time.h>
char *asctime_r(ptr, s);
struct tm *ptr;
char *s;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `asctime_r()` converts the broken-down time given in the tm structure pointed to by `ptr` to an ASCII string pointed to by `s`, in the following format:

DAY MON day hour:min:sec YEAR

This function is the re-entrant replacement for the Draft Proposed ANSI C function `asctime()`. Instead of writing the result of the conversion into a character array in the static data area, a pointer to the resulting string is passed to the function from the caller. The programmer is responsible for the allocation of sufficient memory to contain the full string.

Return values:.... `asctime_r()` returns a pointer to the ASCII string denoting the calendar time given in the `tm` structure pointed to by `ptr`.

Notes and remarks: In contrast to the string which is built by `asctime()`, the format of the string resulting from `asctime_r()` does not contain a newline character.

See also: ....

```
asctime()
time()
time.h
```

asin()

Usage: ....
```
#include <math.h>
double asin(x);
double x;
```

Availability: ..... Draft Proposed ANSI C

Libraries: .... `math.lib` and `t8math.lib`

Description: ..... `asin()` computes the arc sine of `x`. The argument `x` is expected to have the type double.

Return values: ... `asin()` returns the value of the arc sine of `x`.

assert()

Usage: ....
```
#include <assert.h>
void assert(expression)
```

Availability: ...... Draft Proposed ANSI C

Description: ..... The macro `assert()` takes as its argument a value of any scalar type; if that value is zero and the macro NDEBUG is *not* defined, `assert()` will print a diagnostic message to the `stdout` stream and terminate the program by calling `exit()`. If the macro NDEBUG is defined, `assert()` is disabled and the expression is not evaluated.

Return values:.... none

atan()

Usage: ....

```
#include <math.h>
double atan(x);
double x;
```

Availability: ..... Draft Proposed ANSI C

Libraries: .... `math.lib` and `t8math.lib`

Description: ..... `atan()` computes the arc tangent of `x`. The argument `x` is expected to have the type double.

Return values: ... `atan()` returns the value of the arc tangent of `x`.

atan2()

Usage: ....
```
#include <math.h>
double atan2(y, x);
double y, x;
```

Availability: ...... Draft Proposed ANSI C

Libraries: .... `math.lib` and `t8math.lib`

Description: ..... `atan2()` computes the angle between the vector (x,y) and the positive x-axis. The result is in radians and lies in the range (-pi, pi]. The arguments are expected to have type double.

Return values: .... `atan2()` returns the value of the computed angle.

atexit()

Usage: ....
```
#include <stdlib.h>
int atexit(fptr);
void (*fptr)();
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `atexit()` sets up the function pointed to by `fptr` to be called without arguments at normal program termination. Consecutive calls to `atexit()` will cause the indicated functions to be listed and executed at program termination in the reverse order. The maximum number of functions that can be listed in this way is 32.

Notes and remarks: `atexit()` does the same as `onexit()`.

Return values: ... `atexit()` returns zero if registration of the indicated function was successful. Otherwise, a nonzero value is returned.

See also: ....
`exit()`
`onexit()`

Example: . . . .

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

void func1()
{
   printf("Executing first exit handler\n");
}

void func2()
{
   printf("Executing second exit handler\n");
}

void main()
{
   atexit(func1);
   atexit(func2);
   printf("Now exiting...\n");
}
```

_ato()

Usage: ....

```
#include <stdlib.h>
long _ato(string, control, output);
char   *string;
int  control;
char   **output;
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: ..... `_ato()` is a generic ASCII-to-number conversion function. `string` points to a character array in which the string to be converted is stored.

`control` specifies the conversion type and consists of a bitwise or of the radix with additional bitfields. The radix may be in the range 2-36. A radix less than 2 results in no conversion taking place and an empty string being returned. A radix greater than 36 results in the possibility of non-alphanumeric characters being inserted.

The bitmasks given in `stdcnv.h` (which is included in `stdlib.h`), further specify the conversion :

`_CV_LONG` : Output is to be considered 64 bits long. If not specified, conversion does not proceed beyond 32 bits.

`_CV_UNSIGNED` : Perform unsigned conversion.

If output is not the `NULL` pointer, a pointer to the non-converted remainder of the string is returned herein.

Return values: ... `_ato()` returns the value of the converted string, represented as an unsigned long integer.

See also: ....

```
ato..()
strtol()
```

ato..()

Usage: ....

```
#include <stdlib.h>
[unsigned] [long] int ato..(string);
char *string;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `ato..()` indicates a collection of ASCII-to-number conversion facilities, which are all implemented as macro's expanding to a call to the generic conversion routine `_ato()`. The following gives a list of the defined macro's and their functions. The return value of `_ato()` is cast to the type expected.

atoi	decimal ASCII string to integer
atol	decimal ASCII string to long
atox	hexadecimal ASCII string to integer
atoxl	hexadecimal ASCII string to long
atou	decimal ASCII string to unsigned integer
atoul	decimal ASCII string to unsigned long
atoo	octal ASCII string to integer
atool	octal ASCII string to long
atob	binary ASCII string to integer
atobl	binary ASCII string to long

Return values:.... All `ato..()` functions return the converted value in the type expected.

See also: .... `_ato()`
`stdcnv.h`

atod()

Usage: ....

```
#include <stdlib.h>
double atod(s);
char *s;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `atod()` converts the initial portion of the string pointed to by `s` to a floating-point number of type double.

Return values: ... `atod()` returns the converted double value.

Notes and remarks: `atod()` is implemented as a macro expanding to a `strtod()` call, and therefore has the same functionality.

`atod()` is defined identically to `atof()`, which is also included in the `stdcnv.h` header file.

See also: ....

`strtod()`
`stdcnv.h`

atof()

Usage: ....
```
#include <stdlib.h>
double atof(s);
char *s;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `atof()` converts the initial portion of the string pointed to by `s` to a floating-point number of type double.

Return values:.... `atof()` returns the converted double value.

Notes and remarks: `atof()` is implemented as a macro expanding to a `strtod()` call, and therefore has the same functionality.

`atof()` is also defined in `stdcnv.h` which is a special header file available in the Par.C System for all conversion facilities.

`atof()` is identical to `atod()`, which is also included in the `stdcnv.h` header file.

See also: .... `strtod()`
`stdcnv.h`

atoi()

Usage: ....

```
#include <stdlib.h>
int atoi(s);
char *s;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `atoi()` converts the initial portion of the string pointed to by `s` to a decimal number of type (signed) integer.

Return values: ... `atoi()` returns the converted integer value.

Notes and remarks: In the Par.C System `atoi()` is defined as a macro expanding to a call to the generic ASCII to integer conversion routine `_ato()`, which is also described in this chapter. `atoi()` is also included as a function in the library `std.lib`.

See also: ....

```
_ato()
stdcnv.h
```

atol()

Usage: ....
```
#include <stdlib.h>
long int atol(s);
char *s;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `atol()` converts the initial portion of the string pointed to by `s` to a decimal number of type (signed) long integer.

Return values:.... `atol()` returns the converted long integer value.

Notes and remarks: In the Par.C System `atol()` is defined as a macro expanding to a call to the generic ASCII to integer conversion routine `_ato()`, which is also described in this chapter. `atol()` is also included as a function in the library `std.lib`.

See also: .... `_ato()`
`stdcnv.h`

BEFORE()

Usage: ....

```
#include <time.h>
int BEFORE(T1, T2)
time_t T1, T2;
```

Availability: ..... Par.C System

Description: ..... The macro `BEFORE()` compares two times and reports if `T1` precedes `T2`.

Return values: ... `BEFORE()` returns zero if `T1` does not precede `T2`, otherwise a non-zero value is returned.

See also: .... `AFTER()`

bltoa()

Usage: ....
```
#include <stdcnv.h>
int bltoa(num, str)
long int num;
char *str;
```

Availability: ...... Par.C System

Description: ..... The macro `bltoa()` converts a long integer (64 bits) to its ASCII representation in binary notation, and stores the result in the string pointed to by `str`. The string should be sufficiently long to store the result of the conversion.

Return values:.... The length of the converted string.

See also: .... `_toa()`
`stdcnv.h`

bsearch()

Usage: ....

```
#include <stdlib.h>
char *bsearch(key, base, n, size, compare);
char *key, *base;
size_t n, size;
int (*compare)(p1, p2);
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `bsearch()` takes `key` as a pointer to the object to search for, `base` as a pointer to the array to search, `n` as the number of elements in the array, `size` as the size of an element and `compare` as a pointer to the comparison function to be used. The array must be in ascending order according to the comparison function.

`compare()` takes two pointers to array elements as arguments. If the object pointed to by the first argument is less than, equal to or greater than the object pointed to by the second argument, `(*compare)()` returns a value less than, equal to or greater than zero respectively.

Return values: ... `bsearch()` returns a pointer to the matching object, or a NULL pointer if the object was not present in the array.

Notes and remarks: `bsearch()` is an implementation of a binary search. Therefore, the elements must be in ascending order. `qsort()` can be used to sort the array in ascending order using the same comparison function.

See also: .... `qsort()`

Example: ....

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

#define MAXLISTLEN 20 /* maximum length of the list */

typedef struct {
   int number;
   char name[20];
   int birthdate;
} element;

element list[MAXLISTLEN]; /* array to search */

int ReadFromFile(filename,listp) /* read elements */
char *filename;
element *listp;
{
   FILE *listfile;
   int actualnumber = 0;

   listfile = fopen(filename,"r");
   if(listfile == NULL)
   {
       fprintf(stderr,"Failed to open %s\n",filename);
       exit(1);
   }

   while(actualnumber < MAXLISTLEN )
   {
       fscanf(listfile,"%d",&listp->number);
       if(listp-number == -1) break; /* end of list */
       fscanf(listfile,"%s %d\n",
           listp->name,
           &listp->birthdate );
       listp++;
       actualnumber++;
   }

   fclose(listfile);
   return actualnumber; /* the number of elements read */
}
```

```
int CompareNameNumber(s1,s2) /* compare name/number */
element *s1,*s2;
{
   int retval;
   retval=strcmp(s1->name,s2->name);
   if (retval != 0)
       return (retval);
   else
       return (s1->number - s2->number);
}

void main()
{
   element search, *found;
   int listlen;

   listlen = ReadFromFile("bsearch.lst",list);
   qsort((char*)list,listlen,sizeof(element),
       CompareNameNumber);

   /* set element to search for */
   search.number=10;
   strcpy(search.name,"Margareth");

/**** find it in list and print other fields *****/
   found=(element *)bsearch(&search,list,listlen,
       sizeof(element), CompareNameNumber);
   if (found != NULL){
       printf("Search element found\n");
       printf("Number:     %20d\n",found->number);
       printf("Name:       %20s\n",found->name);
       printf("Birthdate: %20d\n",found->birthdate);
   }else
       printf("Search element not found in list\n");
}
```

btoa()

Usage: ....

```
#include <stdcnv.h>
int btoa(num, str)
int num;
char *str;
```

Availability: ...... Par.C System

Description: ..... The macro `btoa()` converts the integer num to its ASCII representation in binary notation, and stores the result in the string pointed to by `str`. The string should be sufficiently long to store the result of the conversion.

Return values:.... The length of the converted string.

See also: .... `_toa()`
`stdcnv.h`

calloc()

Usage: ....
```
#include <stdlib.h>
#include <stddef.h>
char *calloc(n, size);
size_t n;
size_t size;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `calloc()` allocates a block of storage space for an array of `n` objects, each with a space of size bytes. Each object is initialised to contain the value 0.

Return values:.... `calloc()` returns a pointer to the start (lowest byte address) of the allocated space. A `NULL` pointer will be returned in case the space can not be allocated, or in case `n` or `size` is zero.

See also: ....
```
malloc()
smalloc()
free()
realloc()
```

ceil()

Usage: ....
```
#include <math.h>
double ceil(x);
double x;
```

Availability: ...... Draft Proposed ANSI C

Libraries: .... `math.lib` and `t8math.lib`

Description: ..... `ceil()` computes the smallest integer not less than `x` and converts it to a double.

Return values: .... `ceil()` returns a double with zero fractional part thus representing an integer not less than `x`.

See also: .... `floor()`
`modf()`

Example: ....
```
#include <math.h>
#include <stdio.h>

void main()
{
   double  x= 2.34,
           y=-4.50,
           z= 1.00;

   printf("%f  %f  %f\n", x, y, z );
   printf("%f  %f  %f\n", ceil(x), ceil(y), ceil(z) );
}
```

Execution of this program will cause the following values to be displayed:
```
2.340000  -4.500000  1.000000
3.000000  -4.000000  1.000000
```

clearerr()

Usage: ....

```
#include <stdio.h>
void clearerr(stream);
FILE *stream;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `clearerr()` clears the end-of-file and error indicators for the stream pointed to by stream. These indicators are otherwise only cleared when the file is opened or when `fflush()`, `fseek()` or `rewind()` are called for the indicated stream.

Return values: .. none

See also: ....

`fflush()`
`fseek()`
`rewind()`

_ClrHaltErr()

Usage: ....
```
#include <system.h>
void    _ClrHaltErr();
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `_ClrHaltErr()` clears the transputer's halt-on-error flag.

Return values: .... none

See also: .... `_SetHaltErr()`

clock()

Usage: ....

```
#include <time.h>
clock_t clock();
```

Availability: ..... Draft Proposed ANSI C

Library:......... `std.lib`

Description: ..... `clock()` can be used to determine the processor time used by the running program up to the moment this function is called. The transputer clock is set to `MostNeg` just before starting the program. The difference between the return value and `MostNeg` divided by `CLOCKS_PER_SEC` gives an approximation to the used time in seconds.

Return values: ... `clock()` returns the value of the internal transputer clock.

Notes and remarks: Since the transputer contains two processor clocks, one for each level of priority, which run at different speeds, the values obtained by `clock()` do not give a time measurement in fractions of seconds immediately. To obtain a time measurement in seconds one should divide the return value of `clock()` by `CLOCKS_PER_SEC`, which is replaced by a priority-dependent value at runtime (15625 for low, 1000000 for high priority).

See also: .... `time.h`

cos()

Usage: ....
```
#include <math.h>
double cos(x);
double x;
```

Availability: ...... Draft Proposed ANSI C

Libraries: .... `math.lib` and `t8math.lib`

Description: ..... `cos()` computes the cosine of `x`. The argument `x` is expected to have the type double.

Return values: .... `cos()` returns the value of the cosine of `x`.

cosh()

Usage: ....

```
#include <math.h>
double cosh(x);
double x;
```

Availability: ..... Draft Proposed ANSI C

Libraries: .... `math.lib` and `t8math.lib`

Description: ..... `cosh()` computes the hyperbolic cosine of `x`. The argument `x` is expected to have the type double.

Return values: ... `cosh()` returns the value of the hyperbolic cosine of `x`.

ctime()

Usage: ....
```
#include <time.h>
char *ctime(*timeptr);
time_t *timeptr;
```

Availability: ...... Draft Proposed ANSI C

Library: .... macro defined in `time.h`

Description: ..... `ctime()` converts the calendar time pointed to by `timeptr` to local time in the form of a string. `ctime()` is equivalent to `asctime(localtime())` and in the Par.C System it is implemented as a macro expanding to that call.

Return values: ... `ctime()` returns a pointer to an ASCII string denoting the local time calculated on the basis of the calendar time given in the `tm` structure pointed to by `timeptr`.

See also: ....
```
asctime_r()
localtime_r()
time()
time.h
```

DateToInt()

Usage: ....
```
#include <time.h>
int DateToInt(day, month, year);
int day, month, year;
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: .... `DateToInt()` converts a date to an integer in such a way that when any two valid dates are converted to integers by calling this function and the resulting integers are subtracted, the result gives the exact number of days between those two dates. When the value of `year` is less than 300, 1900 is added before conversion. The value of `month` should be in the range 0-11 (*not* 1-12), so January is represented by 0, December by 11.

No parameter check is done. If any parameter is out of range, or they form an unconsistent date, the result is unpredictable. `yday()` can be used to check the validity of a certain date.

Multiplication with 86400 (= 24*60*60) and adding a number of seconds since 00:00:00 will yield a value, which can be recognized and converted as a valid time by the routines `localtime()`, `localtime_r()`, `gmtime()` and `gmtime_r()`.

Return values: .. `DateToInt()` returns a linear date, measured from the first of March of 1972.

See also: ....
```
IntToDate()
yday()
localtime()
localtime_r()
gmtime()
gmtime_r()
```

delay()

Usage: ....
```
#include <math.h>
void delay(t)
unsigned int t;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... The function `delay()` suspends execution of the current process for `t` milliseconds, during which time another process can be scheduled and executed.

Return values: .... none

Deschedule()

Usage: ....
```
#include <transp.h>
void Deschedule();
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: ..... `Deschedule()` causes the current process to be added to the back of the process queue. The next process will be scheduled.

Return values: ... none

difftime()

Usage: . . . .
```
#include <time.h>
double difftime(time2, time1);
time_t time2, time1;
```

Availability: . . . . . . Draft Proposed ANSI C

Library: . . . . `std.lib`

Description: . . . . . `difftime()` computes the difference in seconds between the calendar times given in both arguments, and returns the result as a double.

Return values: . . . . `difftime()` returns the value in seconds of `time2` - `time1`, represented as a double.

Dir()

Usage: ....

```
#include <stdio.h>
DIR *Dir(search);
DIR *search;
```

Availability: ..... Par.C System

Library: .... `io.lib`

Description: .... `Dir()` searches for the (next) file, that matches the file descriptor as given to `DirInit()` when the search was initialised. After returning, the structure at which `search` is pointing, contains information on the attributes of the file that was found. When no (next) matching file could be found, `Dir()` returns a `NULL` pointer and de-allocates the memory used for the `DIR` structure.

In the `DIR` structure the string members are padded with null characters. The structure is defined in `<stdio.h>`. The meaning of the members (or fields) depends on the filesystem of the host the search is directed to. If this system does not support for instance access codes or version numbering, those fields will be zero. In file systems that support filenames longer than 32 characters, the filename will be truncated.

Return values: .. `Dir()` returns its argument when a matching file is found. A `NULL` pointer is returned when no matching file could be found.

Notes and remarks: The `DIR` structure is filled with host file information, for the exact mapping of these bits compare `DIR` in stdio.h with comparable information in your host manuals.

Extension fields are considered part of the filename.

See also: . . . . `DirInit()`

Example: . . . .

```
#include <stdio.h>
int main()
{
   DIR *DS;
   char*filename = "DRIVER.*";

   if ( !(DS = DirInit(filename)) )
   {
       fprintf(stderr,"No directory info supported\n");
   }
   else
       if (!Dir(DS))
       {
           fprintf(stderr,"No match for %s found\n",
                                filename);
       }
       else
           do
           {
               printf("%.12s %d %d %d %d\n",
                        DS->name, DS->type, DS->version,
                        DS->size, DS->access);
           } while (Dir(DS));
}
```

DirInit()

Usage: ....

```
#include <stdio.h>
DIR     *DirInit(filedesc)
char*   filedesc;
```

Availability: ..... Par.C

Library: .... `io.lib`

Description: .... `DirInit()` initiates a file search in a directory on the host file-system and allocates memory for the `DIR` structure.

If the file system does not support directory inquiries then no memory is allocated. To obtain information about files in the host file-system this function should be called before `Dir()`, which actually reads the attributes of a file (a file can also be a sub-directory) and puts that information in the `DIR` structure.

Argument `filedesc` (filedescriptor) is a pointer to a string, containing the search criterion. This search criterion is dependent on the service of the filesystem the search is directed to. Most filesystems allow both filename and extension to be replaced by 'wildcards', which is generally an asterisk (*). Sometimes also part of the filename may be an asterisk, and often question marks are used, denoting that any character may match at that spot. Most times wildcards in directory-specifications are not allowed. As the Par.C filesystem is depending either on a host machine or a user-defined server, one should refer to the appropriate documentation on the host file-system for the exact format of the descriptor.

Return values: .. `DirInit()` returns a pointer to a structure `DIR`, which can be used by `Dir()` to fill that structure. A `NULL` pointer is returned when no directory inquiries are supported.

Notes and remarks: Note that the structure the return value is pointing at, does not contain valid information yet. So even if only one file is to be expected to match (i.e. no wildcards are used) at least one call to `Dir()` is required to obtain the directory information.

See also: .... `Dir()`

dtoa()

Usage: ....
```
#include <stdcnv.h>
int dtoa(d, str)
double d;
char *str;
```

Availability: ...... Par.C System

Description: ..... This macro is an alias for the function `ftoa()`, but with its parameters reversed. It will put the double representation of `d` into `str`.

Return values: ... The length of the converted string.

(*ERROR)()

Usage: ....

```
#include <errno.h>
void    (*ERROR)(errnumber, descr, ...)
int     errnumber;
char    *descr;
```

Availability: ..... Par.C System

Libraries: .... `io.lib` and `noio.lib`

Description: .... `ERROR` is a pointer to an error function, which can be replaced to point to an user supplied function. All library routines, in case of an error, call `(*ERROR)()` for error handling, where needed. By default, `ERROR` points to an error function, which is defined differently in `io.lib` and `noio.lib`. In `io.lib` the default error function sends a special error message to the server on the host which causes the text :

```
"<programname> - *ERROR* : <error description>"
```

to be printed to the standard error stream on the host.

In noio.lib the default error function only sets the transputer hardware error flag.

The argument `errnumber` contains the type of error and can be processed by the `strerror()` function or can be looked up in the `errno.h` file. Argument `descr` has the same form and function as the format string in `printf()`. Besides `errnumber` and `descr` there can be more arguments on the stack, see the example on how to process them.

Return values: ... none

Notes and remarks: Variable argument lists can be implemented using the facilities described in `vararg.h` (see the description of header files at the beginning of this chapter).

See also: .... `errno.h`
`strerror()`
`matherr()`
`perror()`

Example: ....

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

void yourErrFunc( errno, descr )
int errno;
char *descr;
{
   va_list ap;
   va_start( ap, descr );

   fprintf( stderr, "Got error : %x    ", errno );
   vfprintf(stderr, descr, ap );
   fprintf(stderr, "\nwhich means: %s\n", strerror(errno));
}

void main()
{
   FILE *nofile;

   ERROR = yourErrFunc; /* Plug in user error handler */

   nofile = fopen("nonexist","r");
       /* Open a file in read mode */
   fprintf(nofile,"This text is never written\n");
       /* Force an error */
}
```

exit()

Usage: ....
```
#include <stdlib.h>
void exit(status);
int status;
```

Availability: ..... Draft Proposed ANSI C

Library:......... `std.lib`

Description: ..... `exit()` causes the program to terminate, after first calling all functions registered by `atexit()` in the reverse order of registration. Then all open output streams are flushed and all open streams are closed. Finally control is returned to the host environment by emitting an exit code to the server running on the host system. The value of `status` is returned, and signifies successful or unsuccessful termination of the program.

Return values: ... none

See also: ....
`_exit()`
`atexit()`

_exit()

Usage: . . . .
```
#include <stdlib.h>
void _exit(status);
int status;
```

Availability: . . . . . . Par.C System

Library: . . . . `std.lib`

Description: . . . . . `_exit()` causes the program to terminate, without calling all functions registered by `atexit()` and without deactivating the file system. Control is returned to the host environment by emitting an exit code to the server running on the host system. The value of `status` is returned, and signifies successful or unsuccessful termination of the program.

Return values: . . . . none

See also: . . . .
```
exit()
atexit()
```

exp()

Usage: ....

```
#include <math.h>
double exp(x);
double x;
```

Availability: ..... Draft Proposed ANSI C

Libraries: .... `math.lib` and `t8math.lib`

Description: ..... `exp()` computes the exponential function of `x`. The argument `x` is expected to have the type double.

Return values: ... `exp()` returns the exponential value of `x`.

fabs()

Usage: . . . .
```
#include <math.h>
double fabs(x);
double x;
```

Availability: . . . . . . Draft Proposed ANSI C

Libraries: . . . . `math.lib` and `t8math.lib`

Description: . . . . . `fabs()` computes the absolute value of the floating-point number `x`.

Return values: . . . . `fabs()` returns the absolute value of `x`.

fclose()

Usage: ....
```
#include <stdio.h>
int fclose(stream);
FILE *stream;
```

Availability: ..... Draft Proposed ANSI C

Library:......... `io.lib`

Description: ..... `fclose()` will cause the stream pointed to by the only argument to be flushed and the associated file to be closed. If a buffer was in use it is emptied by writing unwritten characters or discarding unread characters. The buffer is deallocated if it was allocated automatically. The stream is disassociated from the closed file.

Return values: ... `fclose()` returns zero if the stream was succesfully closed, or `E_CANTCLOSE` if errors occurred or if the stream was already closed.

See also: .... `errno.h`

feof()

Usage: . . . .
```
#include <stdio.h>
int feof(stream);
FILE *stream;
```

Availability: . . . . . . Draft Proposed ANSI C

Library: . . . . `io.lib`

Description: . . . . . `feof()` tests whether the end-of-file indicator for the stream pointed to by `stream` is set or not. The indicator is not cleared.

Return values: . . . . `feof()` returns a nonzero value if the end-of-file indicator for the specified stream is set.

See also: . . . . `ferror()`

ferror()

Usage: ....
```
#include <stdio.h>
int ferror(stream);
FILE *stream;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `ferror()` tests whether the error indicator for the stream pointed to by `stream` is set or not. The indicator is not cleared.

Return values: ... `ferror()` returns a nonzero value if the error indicator for the specified stream is set.

See also: .... `feof()`

fflush()

Usage: ....
```
#include <stdio.h>
int fflush(stream);
FILE *stream;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `fflush()` will cause any unwritten (buffered) data for the stream pointed to by `stream` to be written to the file or device associated to the stream. If `stream` points to an input stream the effect of the call will be the undoing of any preceding `ungetc()` calls on the stream.

Return values: .... `fflush()` returns zero if the call was succesful; a nonzero value will only be returned in case of write errors occurring.

See also: .... `ungetc()`

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

char buff[512];

void main()
{
   setvbuf(stdout,buff,_IOFBF,512); /* full buffering */
   printf("Hello there\n");
   printf("This is the last line we were flushing\n");
   fprintf(stderr,"(stderr) Now flushing stdout\n");
   fflush(stdout);
}
```

fgetc()

Usage: ....

```
#include <stdio.h>
int fgetc(stream);
FILE *stream;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `fgetc()` reads the next available character from the stream pointed to by `stream`.

Return values: ... `fgetc()` returns a positive integer representing the character read from the input stream. If an end-of-file marker is encountered, the end-of-file indicator is set and EOF is returned. If an error occurs while reading, the error indicator is set and EOF is returned.

See also: ....

`getc()`
`getchar()`

fgets()

Usage: ....
```
#include <stdio.h>
char *fgets(s, n, stream);
char *s;
int n;
FILE *stream;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `fgets()` reads at most `n-1` characters from the stream pointed to by `stream` and copies these characters to the string pointed to by `s`. A terminating null character is appended to the string.

Return values: .... `fgets()` returns `s` if reading was successful. If an end-of-file marker is encountered or if an error occurs while reading the `NULL` pointer is returned.

Notes and remarks: If `fgets()` was unsuccessful the contents of the character array pointed to by `s` depend on the number of characters read into the array so far. This implies that the remaining array may not contain a terminating null character anymore, and therefore cannot be treated as a string.

See also: .... `gets()`

filerr()

Usage: ....

```
#include <stdio.h>
int filerr(fp)
FILE *fp;
```

Availability: ..... Par.C System

Description: .... The macro `filerr()` tests the status of the stream pointed to by `fp`.

Return values: .. The `filerr()` macro returns the error code of the last file I/O command for this stream. This code is the same as the one found in the global variable `errno`.

floor()

Usage: . . . .

```
#include <math.h>
double floor(x);
double x;
```

Availability: . . . . . . Draft Proposed ANSI C

Libraries: . . . . `math.lib` and `t8math.lib`

Description: . . . . . `floor()` computes the largest integer, which is not greater than `x`.

Return values: . . . . `floor()` returns the largest integer not greater than `x`, represented as a double.

See also: . . . . `ceil()`
`modf()`

fmod()

Usage: ....
```
#include <math.h>
double fmod(x, y);
double x, y;
```

Availability: ..... Draft Proposed ANSI C

Libraries: .... `math.lib` and `t8math.lib`

Description: ..... `fmod()` computes the floating-point remainder of `x/y`, i.e. the value `z` such that `z` and `x` have equal sign, the absolute value of `z` is less than the absolute value of `y` and `x=z+k*y` for some integer `k`.

Return values: ... `fmod()` returns `z`, the floating-point remainder of `x/y`.

fopen()

Usage: ....
```
#include <stdio.h>
FILE *fopen(filename, mode);
char *filename, *mode;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `fopen()` will cause the file with name `filename` to be opened in the mode indicated by `mode`, and a stream to be associated to the opened file. A pointer to the associated stream is returned. The string pointed to by `mode` should start with one of the following characters:

r	open file for reading
w	create file for writing, or truncate existing file to zero length
a	create file for writing, or open existing file for writing at end-of-file. To this first character of `mode`, one or both of the following characters may be added:
b	open a binary file (instead of the default text file mode)
+	open the file for reading and writing. Opening a file in the read mode ('r' as first character of `mode`) fails if the file does not exist or cannot be read. Opening a file in the append mode ('a' as first character of `mode`) causes all subsequent writes to the file to be forced to the current end-of-file.

Return values: .... `fopen()` returns a pointer to the stream associated with the file opened. If opening of the file with name `filename` in the mode indicated by `mode` was unsuccessful, the `NULL` pointer is returned.

Notes and remarks: When a file is opened in update mode (+ as second or third character of `mode`) both input and output are possible on the file. However, output must be followed by a call to `fflush()`, `fseek()` or `rewind()`, before the next input is read from the file. Also, input may not be followed directly by output without an intervening call to the `fflush()` or `fseek()`

function, unless an end-of-file has been encountered in the input. The type of buffering for streams opened via `fopen()` is set to full buffering for binary files and line buffering for text files. This type of buffering can be changed using `setvbuf()`.

See also: ....

```
fseek()
rewind()
setvbuf()
```

Example: ....

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

FILE *fp;
char buff[128];
int len;

void main()
{
   if ((fp = fopen("trash","w")) == NULL) {
       printf("could not open 'trash'(w)\n");
       exit(1);
   }
   fputs("Hai the ho\n",fp);
   fputs("Will this work?\n",fp);
   fclose(fp);

   if ((fp = fopen("trash","rb")) == NULL) {
       printf("could not open 'trash'(rb)\n");
       exit(1);
   }
   len = fread(buff,1,128,fp);              /* read text */
   buff[len] = '\0';            /* append terminating null */
   printf("read %d bytes : %s",len,buff);
   fclose(fp);
}
```

fprintf()

Usage: ....

```
#include <stdio.h>
int fprintf(stream, format, ...);
FILE *stream;
char *format;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `fprintf()` writes output to the stream pointed to by `stream` , in the format indicated in the format string pointed to by `format` , using the optional arguments following the format string.

The format string may contain zero or more ordinary characters which are copied unchanged to the output stream and zero or more conversion specifications, resulting in one or more subsequent arguments being converted to printing format and written to the output stream. Each conversion specification is preceded by the character `%`. Writing the character `%` to the output stream is achieved by putting `%%` in the format string.

format specifiers

After the `%` character, the following characters may appear in sequence:

- Zero or more flags modifying the meaning of the conversion specification following the flags;
- An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than this minimum field width, padding will be used to fill the remaining characters in the field. Padding is done according to the flags preceding the conversion specification. Default is padding with spaces and right-adjusted. If the converted value takes up more space than the specified field width, the field is expanded to contain the complete conversion result.

The field width indicator may be replaced with an asterisk (`*`) instead of a digit string. In this case the next argument in the list is taken to indicate the field width.

- An optional precision indication, giving the minimum number of digits to appear for integer conversions, the number of digits to appear after the decimal-point character for floating point type conversions (except when g or G is specified), the maximum number of significant digits for floating point conversions when g or G is specified, the maximum number of characters to be written for string conversions, or a repeat count for character conversions.

The precision indication consists of a decimal-point followed by a digit string. If the digit string is left out, the precision indication is set to zero. The digit string may be replaced with an asterisk (*), causing the next argument to be taken as precision indicator. A negative value of this argument will be taken as if the precision indicator was missing, and therefore cause the precision to be set to the default value for the type concerned.

The amount of padding specified by the precision overrules the specified field width in the case of insufficient positions being available to print the argument with the precision specified.

- An optional indicator 'h' for short integers, or 'l' for long integers. If one of these indicators is specified with a conversion specifier to which it cannot apply, the indicator is ignored.
- A conversion specifier, indicating the type of the argument to be converted to printing format.

flags

The flag characters and their meanings are given below:

-	The result of the conversion will be left-adjusted within the field. The default mode is right-adjustment.
+	The result of the conversion of a signed type will be printed with a plus or minus sign.
=	The result of the conversion will be centered within the field. (non-ANSI C feature)
<space>	If the first character of the conversion of a signed type is not a sign, a space will be printed preceding the result of the conversion. If both space and plus sign appear as flags, the space flag will be ignored.
#	The result of the conversion of an octal number will be preceded by a zero. The result of the conversion of a nonzero hexadecimal number will

be preceded by "0x" or "0X". The result of the conversion of a floating point number will include a decimal point character.

0	Zero padding will be used rather than space padding. However, zero padding is overruled when also specifying "-" for left-adjustment.

conversion specifiers

The conversion specifiers and their meanings are given below

d,i	The argument is expected to be an integer and is printed in signed decimal notation.
u	The argument is expected to be an integer and is printed in unsigned decimal notation.
b	The argument is expected to be an integer and is printed in unsigned binary notation.
o	The argument is expected to be an integer and is printed in unsigned octal notation.
x,X	The argument is expected to be an integer and is printed in unsigned hexadecimal notation. 'x' and 'X' specify lower and upper case for the hexadecimal digits a through f.
f	The argument is expected to be a double and is printed in signed decimal fixed-point notation. The number of digits appearing behind the decimal-point character is given by the optional precision specifier. The default precision is 6. At least one digit will appear preceding the decimal-point character. If the specified precision is zero and the # flag is not specified, no decimal point will be printed.
e,E	The argument is expected to be a double and is printed in signed decimal floating-point notation with one digit preceding the decimal-point character and an exponent introduced by the character 'e' or 'E'. The number of digits appearing behind the decimal-point character is given by the optional precision specifier. The default precision is 6. The exponent part will always contain at least two digits.
g,G	The argument is expected to be a double and is printed in signed decimal floating-point notation, in the style of the 'f' or 'e' specifiers ('E' in the case of 'G') according to the way which will take up the least positions. The style of 'e' (or 'E') will be used when the exponent resulting from the conversion is less than -4 or greater than the precision indicated.

Trailing zeroes are removed from the result, and a decimal-point character is printed only if it is followed by a digit (unless the "#" flag is set).

c	The argument is expected to be an integer and is converted to an unsigned char before it is printed as a character.
s	The argument is expected to be a pointer to a string and characters from this string are printed up to, but not including the terminating null character. If the precision is specified, no more than the specified number of characters are printed.
p	The argument is expected to be a pointer, of which the value is converted to a sequence of printable characters in the style of a hexadecimal number (according to the 'x' specifier).
n	The argument is expected to be a pointer to an integer, which is used to write the number of characters, which have been printed to the output stream at the moment the n specifier is encountered. This "conversion" specifier does not cause a conversion to be executed, since the value is not printed to the output stream.

Return values: . . . `fprintf()` returns the number of characters which have been written to the indicated stream, or a negative value if an error has occurred.

Notes and remarks: Using a precision indicator with the "c" conversion causes the character to be printed repeatedly, the number of characters printed being specified by the precesion indicator. This is an extension of the use of the precision indicator as compared with the Draft Proposed ANSI C standard.

The centralise option flag '=' is not part of the Draft Proposed ANSI C standard.

See also: . . . . `printf()`

`sprintf()`

`vprintf()`

`vfprintf()`

`vsprintf()`

fputc()

Usage: ....
```
#include <stdio.h>
int fputc(c, stream);
int c;
FILE *stream;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `fputc()` writes the character `c` to the stream pointed to by `stream`.

Return values:.... `fputc()` returns a positive integer representing the character written to the stream. If an error occurs while writing, the error indicator is set and `EOF` is returned.

Notes and remarks: The macro `putc()` expands to a considerable amount of code, but is faster than a call to `fputc()`.

See also: .... `putc()`

`putchar()`

fputs()

Usage: ....

```
#include <stdio.h>
int fputs(s, stream);
char *s;
FILE *stream;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `fputs()` writes the string pointed to by `s` to the stream pointed to by `stream`.. The terminating null character is not written.

Return values: ... `fputs()` returns a nonzero value if the string has been successfully written. Otherwise, zero is returned.

See also: .... `puts()`

fread()

Usage: ....

```
#include <stdio.h>
size_t fread(dest, size, n, stream);
char * dest;
size_t size , n;
FILE *stream;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `fread()` reads up to `n` items, each of size `size` from the stream pointed to by `stream` into the array pointed to by `dest`. If a read error occurs or an end-of-file marker is encountered, execution is aborted. A call to `ferror()` or `feof()` can be used to determine which error has occurred.

Return values: .... `fread()` returns the number of items of the array which have been read successfully. This number may be less than the number specified by `n`, in which case an error has occurred. If `size` or `n` is zero, the returned value is zero and the contents of the array and the state of the stream remain unchanged.

Notes and remarks: If an EOF is encountered before `n` elements have been read, the return value gives the number of elements successfully read, and the next element will be partially filled.

See also: .... `fwrite()`

`feof()`

`ferror()`

free()

Usage: ....
```
#include <stdlib.h>
int free(ptr);
char *ptr;
```

Availability: ..... Draft Proposed ANSI C

Libraries: .... `a.lib and b.lib`

Description: ..... `free()` causes the memory block pointed to by `ptr` to be deallocated. If the argument does not match a valid memory block pointer (such as obtained by a call to one of the memory allocation functions) a non-zero value is returned to indicate an error.

Return values: ... `free()` returns a non-zero value if `ptr` was found to indicate an invalid address. When the block pointed to by `ptr` was succesfully deallocated, a zero value is returned.

See also: .... `malloc()`

`smalloc()`

frexp()

Usage: ....
```
#include <math.h>
double frexp(x, exp);
double x;
int *exp;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `math.lib` and `t8math.lib`

Description: ..... `frexp()` breaks a floating-point number into a normalized fraction and a power of 2. The argument `x` is expected to have the type double. The power of 2 is stored in an integer pointed to by `exp`.

Return values:.... `frexp()` returns the value `y`, such that `y` is a double with magnitude in the interval [0.5, 1) or zero, and `x` equals $y * 2^{exp}$. If `x` is zero, both parts of the result are zero.

See also: .... `ldexp()`

`pow2()`

Example: ....
```
#include <math.h>
#include <stdio.h>

void main()
{
   double x = 3.1415e214, y;
   int exp;

   y = frexp(x, &exp);
   printf("x : %le\ny : %f  exp : %d\n", x, y, exp);
   printf("y * 2 ^ exp : %le\n", y*pow2((double)exp));
}
```

fscanf()

Usage: ....
```
#include <stdio.h>
int fscanf(stream, format, ...);
FILE *stream;
char *format;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `fscanf()` reads input from the stream pointed to by `stream`, which is converted according to the format indicated in the format string pointed to by `format`, using the optional arguments following the format string to receive the input.

The format string may contain one or more whitespace characters, a number of ordinary characters (excluding the % character) and a number of conversion specification directives, each introduced with the % character. Whitespace characters are matched with any sequence of whitespace characters, other characters must literally match for conversion to continue. When a % is encountered, a conversion from input data will be performed.

format specifiers

Following the % character the following characters may appear in sequence:

- An optional asterisk (*) indicating that assignment of the following input should be suppressed.
- An optional decimal integer specifying the maximum field width. This denotes the maximum number of characters, that will be read.
- An optional character indicating the size of the receiving object. The indicator 'h' causes a short integer (16 bit) or float (32 bit) to be expected; 'l' indicates a long integer (64 bit) or double (64 bit). The default size of the assignment is 32 bits for integer as well as for float conversion. If one of these indicators is specified with a conversion specifier to which it cannot apply, the indicator is ignored.
- A conversion specifier, indicating the type of conversion to be applied to the input before assignment to an object.

`fscanf()` executes the directives given in the format string one by one, and returns when a directive fails. Failures may result from an input error or from an error in the attempt to match the input with the specified conversion type. If an error is caused by a mismatch in the input stream, the character causing the error remains unread.

An ordinary character in the format string causes the next character to be read from the input stream. If the character differs from the input, the directive fails and the character remains unread.

A whitespace in the format string will cause characters to be read up to and excluding the first following non-whitespace character, or until no more characters can be read.

Each conversion specification is processed as follows:

- Whitespace characters are skipped, except with the '[', 'c', or 'n' specifiers.
- Unless the conversion specifier is 'n', an input item is read from the stream. The input item is defined as the longest sequence of input characters (up to a possibly specified maximum field width), which is an initial subsequence of the specified matching sequence. If the length of the input item is zero, execution of the directive fails.
- The input item is converted to the type indicated by the conversion specifier and unless the * character is specified, the result of the conversion is assigned to the object pointed to by the next argument in the argument list.

conversion specifiers

The conversion specifiers and their meanings are given below:

d The input is matched with an optionally signed decimal integer and the conversion result is assigned to the integer the next argument points to.

i As d, but when the number input is preceeded by 0, as o. When preceeded by 0x, as x.

When a precision is specified, it represents a radix, e.g. `%.13i` reads a number with radix 13 (see also : extra features added after the notes and remarks).

o The input is matched with an optionally signed octal integer and the conversion result is assigned to the integer the next argument points to.

u

The input is matched with an unsigned decimal integer and the conversion result is assigned to the unsigned integer the next argument points to.

x The input is matched with an unsigned hexadecimal integer and the conversion result is assigned to the integer the next argument points to.

e,f,g The input is matched with an signed floating-point number. The conversion result is of type float and is assigned to the float or (after a float to double conversion) to the double the next argument points to. Double assignment is forced by a 'l' specifier.

s The input is matched with a sequence of non-whitespace characters. The result is assigned to the string the next argument points to.

[This character should be followed in the format string by a set of characters terminated by the corresponding ']' character, which is treated as the scanset and indicates the characters which will cause copying input characters to continue. The characters are copied to a character array pointed to by the next argument. After the input sequence is terminated, a trailing null character is appended to the character array.

Input of characters will stop either after the optionally specified field width has been reached, after an end-of-file has been found, or after the first character which is not included in the scanset is encountered (this character will not be read). When the first character following the opening '[' character is the negation sign '^', all characters specified are treated as not being in the set, and the occurrence in the input of one of these characters will stop more characters from being read. Note that both the '[' and '^' characters may be included in the scanset since only one specific position of these characters is treated as conversion specification. The ']' character may be included in the scanset only if it is placed immediately after the conversion specification characters, i.e. when the format string reads "[]...]" or "[^]...]". The second ']' character is needed to terminate the scanset.

c The input is matched with a sequence of characters, the number of which is specified by the field width, and these are copied to the character array the next argument points to. If the field width is not specified, one single character is copied. No terminating null character is added to the array. Whitespace characters are treated like all other characters by this conversion.

p

The input is matched with a sequence of characters indicating a pointer value. The format of the expected string should be the same as is produced by the corresponding print facilities for pointer values, namely a hexadecimal number.

n The argument is expected to be a pointer to an integer, which is used to write the number of characters to, which have been read from the input stream at the moment the 'n' specifier is encountered. This "conversion" specifier does not cause a conversion to be executed, since no character is read from the input stream.

% The input character is matched with a single % character, which is not assigned.

Return values:.... `fscanf()` returns the number of input items assigned, or EOF if an error occurred before any conversion has been executed. Note that the use of the %n conversion specifier does not cause a character to be read and will not increment the return value, neither will conversions with the assignment suppression indicator '*' increment the return value.

Notes and remarks: According to the Draft Proposed ANSI C standard, number conversion always succeeds. So for example a hexadecimal conversion encountering '0xPPP' in the input will assign 0 to the associated variable, after which conversion will proceed at 'PPP'.

A number of extra features have been added. They are mentioned here, because they are not part of the Draft Proposed ANSI C standard :

- Extra conversion '%b' for unsigned binary.
- Binary numbers in the input stream may be preceded by '0b' or '0B
- The %i conversion is a general number conversion of any number for which 1 < radix < 37. A precision is allowed with this conversion, which makes its definition : %[*][<width>][.[-]<precision>][h|l]i

The minus sign instructs `fscanf()` to apply signed conversion. The precision defines which radix is used for the conversion. If the radix is not specified by a precision, it is specified by the input as follows (according to ANSI C) : If the number read is preceded by 0x or 0X, unsigned hexadecimal conversion is applied. If the number read is preceded by 0b or 0B, unsigned binary conversion will take place. If the number read is preceded by one or more leading 0's, it is assumed to be unsigned octal. In other cases signed decimal conversion will be performed.

- The hyphen ('-') can be used in sets to define a range, e.g. :

%[A-Za-z] defines all alphabetic characters. This implies two 'escape' possibilities to include a hyphen as a part of the set, namely mentioning it first, or between two characters, of which the first has a higher value than the second.

Examples : .....

Format	Set
`%[1-4]`	`1234`
`%[-+1-9]`	`+-123456789`
`%[0-9-+]`	`+-0123456789`
`%[+--]`	`+'-`

- Draft Proposed ANSI C specifies that on number conversions characters are read until a character that is not a digit in the specified radix system is encountered in the input stream. If the conversion overflows, the result is undefined. The Par.C implementation does give some guarantees in these cases:

As soon as overflow occurs, depending on the calculus and an 'l' or 'h' specifier, the conversion is stopped, but remaining digits are still being read.

Example : format "%lx" and input : "1234567890abcdef123Z" will yield the number 0x1234567890abcdef to be assigned to a long unsigned int, and the input stream being read until 'Z'.

- The conventional implementation of whitespace matching/skipping would be to have any single whitespace character (' ', '\n', '\r', '\t', '\v', '\f') skip characters from the input stream, until a non-whitespace character is encountered. In the Par.C System, three types of whitespace matching/skipping are implemented :

-A space character matches all conventional whitespace characters, namely : space, tab, <CR>, <LF>, <FF>, <VT>. (this is according to the Draft Proposed ANSI C standard).

-A tab matches spaces and tabs (' ', '\t').

-A <CR>, <LF> <VT> or <FF> matches any sequence of <CR>s, <LF>s, <VT>s and/or <FF>s, but no space or tab.

Before a number conversion is performed, only spaces and tabs are skipped, thereby enabling the conversion to stop when encountering a newline. Skipping all whitespace characters before conversion can be achieved by inserting a space in the format string before the conversion specification.

- The <count> field at the "%[" conversion (sets), does have the expected effect. So "%12[abc]" reads at most 12 characters from the input, even if the 13th is also in the set.

See also: .... `scanf()`

fseek()

Usage: ....

```
#include <stdio.h>
int fseek(stream, offset, ref);
FILE *stream;
long int offset;
int ref;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `fseek()` sets the file position indicator for the stream pointed to by `stream` to point to `offset` characters from the point referred to via the code given in `ref`.

If `stream` is associated with a binary file, `offset` is taken as a signed number of characters. If the corresponding file is treated as text file only `0L` is allowed for `offset` and `SEEK_SET` and `SEEK_END` as reference point indicators. This is overruled when the loader is invoked with the "-b" option, which causes all files to be opened as binary files.

`ref` may have one of the following values:

`SEEK_SET`:	indicates the beginning of the file.
`SEEK_CUR`:	indicates the current position of the file pointer.
`SEEK_END`:	indicates the position of the end-of-file marker.

`fseek()` clears the end-of-file indicator and undoes the effect of `ungetc()` calls for the same stream.

Return values: ... `fseek()` returns a nonzero value if a request was made which could not be granted.

See also: .... `ftell()`

Example: . . . .

```
#include <stdio.h>

FILE *fp;
long pos;

void main()
{
   fp = fopen("temp","w+");
   fprintf(fp,"first world in a string\n");
   fseek(fp,0L,SEEK_SET);  /* return to the beginning */
   fprintf(fp,"Hello ");   /* replace 'first' */
   pos = ftell(fp);        /* remember this position */
   fseek(fp,0L,SEEK_END); /* move to the end of the file */
   fprintf(fp,"This is at the end of the file.\n");
   fseek(fp,pos+12,SEEK_SET);
                           /* move to the right position */
   fprintf(fp,"file..\n"); /* replace 'string' */
   fclose(fp);
}
```

ftell()

Usage: ....

```
#include <stdio.h>
long int ftell(stream);
FILE *stream;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `ftell()` returns the current position of the file position indicator for the stream pointed to by `stream`. For a binary file, the return value equals the number of characters measured from the beginning of the file.

Return values: ... `ftell()` returns the current value of the file position indicator. If an error occurred the return value is `(long)(-1)`.

Notes and remarks: The functioning of `ftell()` is only guaranteed for binary files, since the representation of textfiles differs from one host system to the other. However, when the loader is invoked with the "-b" option, texts files will also be treated as binary files, and `fseek()` can be used on them.

See also: .... `fseek()`

ftoa()

Usage: ....
```
#include <stdlib.h>
int    ftoa(s, fp);
char   *s;
double fp;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `ftoa()` converts the floating-point number `fp`, which is expected to have the type double, to its ASCII representation and stores the result of this conversion in the character array pointed to by `s`. The number is represented in the format corresponding to the %G conversion specifier used in `fprintf()` and related functions.

The size of the array pointed to by `s` should be at least 24 characters.

Return values: ... `ftoa()` returns the length of the string pointed to by `s`, resulting from the conversion of `fp` to ASCII.

Notes and remarks: This function is included for compatibility reasons. It is advised to use the format function `sprintf()`, which offers more formatting possibilities.

See also: .... `fprintf()`

`sprintf()`

fwrite()

Usage: ....

```
#include <stdio.h>
size_t fwrite(src, size, n, stream);
char *src;
size_t size, n;
FILE *stream;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `fwrite()` writes up to `n` items, each of size `size` from the array pointed to by `src` to the stream pointed to by `stream`.

Return values: ... `fwrite()` returns the number of items of the array which have been successfully written. This number may be less than the number specified by `n`, in which case an error has occurred.

See also: .... `fread()`

getc()

Usage: ....

```
#include <stdio.h>
int getc(stream);
FILE *stream;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `getc()` reads the next available character from the stream pointed to by `stream`.

Return values: .... `getc()` returns a positive integer representing the character read from the input stream. If an end-of-file marker is encountered, the end-of-file indicator is set and `EOF` is returned. If an error occurs while reading, the error indicator is set and `EOF` is returned.

Notes and remarks: `getc()` is implemented as a macro, expanding to a lot of code, but executing faster than a call to `fgetc()`. When using `getc()`, side effects in `stream` should be avoided.

See also: .... `fgetc()`

`getchar()`

getchar()

Usage: ....
```
#include <stdio.h>
int getchar();
```

Availability: ..... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `getchar()` reads the next available character from the standard input stream.

Return values: ... `getchar()` returns a positive integer representing the character read from the standard input stream. If an end-of-file marker is encountered, the end-of-file indicator is set and `EOF` is returned. If an error occurs while reading, the error indicator is set and `EOF` is returned.

Notes and remarks: `getchar()` is defined as a macro, expanding to `getc(stdin)`.

See also: .... `getc()`

`fgetc()`

getenv()

Usage: ....
```
#include <stdlib.h>
char *getenv(name);
char *name;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `getenv()` causes the environment list on the host system to be searched for a variable matching the string pointed to by `name`. The string referenced by this variable is obtained from the host system and a pointer to this string is returned to the caller.

Return values: .... `getenv()` returns a pointer to a string containing the value associated with `name` on the host system. If the variable is not found, a `NULL` pointer is returned.

Notes and remarks: Since the data area containing the string obtained from the host system is internal to the `getenv()` function, problems arise when a number of concurrent processes call the function because the requested string might be replaced by another string before it is copied. Therefore, a re-entrant version `getenv_r()` has been added in the Par.C System.

See also: .... `getenv_r()`

getenv_r()

Usage: ....
```
#include <stdlib.h>
int getenv_r(value, name);
char   *value, *name;
```

Availability: ..... Par.C System

Library : .... `io.lib`

Description: .... `getenv_r()` causes the environment list on the host system to be searched for a variable matching the string pointed to by `name`. The string referenced by this variable is obtained from the host system and copied to the string pointed to by `value`. The programmer is responsible for allocation of sufficient memory space to contain the returned string.

Return values: ... `getenv_r()` returns zero if the variable has successfully been read. Otherwise, a nonzero value is returned.

See also: .... `getenv()`

Example: ....
```
#include <stdlib.h>
#include <stdio.h>

void main( argc, argv )
int argc;
char *argv[];
{
   char Var[120];

   if( getenv_r( Var, argv[1] ) )
       printf("%s not defined\n", argv[1] );
   else
       printf("%s=%s\n", argv[1], Var );
}
```

_GetFunStack()

Usage: ....

```
#include <system.h>
int _GetFunStack();
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `_GetFunstack()` returns the current size of the stackspace added to the stack when a function has exhausted its stackspace.

The user program may change this value by using `_SetFunstack()`.

This and related routines can be used to optimize program performance by tuning memory usage for stacks. Using too small stacks degrades function calling speed, using too large stacks wastes memory space. See the chapter on memory usage for an explanation on stack handling during runtime.

Return values:.... The current stacksize for functions.

Notes and remarks: Controlling stackspace is global. During parallel processing some construct (e.g. using semaphores while starting processes) should be employed to optimise for just a few function calls.

See also: .... `_GetParStack()`

`_SetFunStack()`

`_SetParStack()`

GetNodeInfo()

Usage: ....

```
#include <system.h>
int GetNodeInfo(Tn, ninfdest);
int Tn;
NODEINFO *ninfdest;
```

Availability: ..... Par.C System

Library:......... `std.lib`

Description: .... `GetNodeInfo()` obtains information concerning the transputer node indicated by `Tn` and assigns this to the structure pointed to by `ninfdest`.

Return values: ... `GetNodeInfo()` returns zero on success and nonzero when the information is not available.

Notes and remarks: Node information is only available on the first transputer in the network (`_Tn == 1`) and if the program was linked with b.lib.

Example: ....

```
#include system.h
NODEINFO myNodeInfo;
GetNodeInfo(Tn, &myNodeInfo);
```

See also: .... `GetSysInfo()`

_GetParStack()

Usage: ....
```
#include <system.h>
int _GetParStack();
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `_GetParStack()` returns the current initial size of the stackspace that will be used for each of the concurrent processes started inside a `par` construct. This only concerns the initial stack allocated for each concurrent process; when any of these needs more workspace during execution (i.e. when recursive function calls are used) use `_SetParStack()` to set a larger initial stack.

See `_GetFunStack()` and `_SetFunStack()` for setting the size of newly allocated stacks for function calls.

Return value: .... The current initial process stacksize.

Notes and remarks: Stacks for the newly created processes and stacks for function calls are mutually independent.

See also: .... `_GetFunStack()`

`_SetParStack()`

`_SetFunStack()`

gets()

Usage: ....
```
#include <stdio.h>
char *gets(s);
char *s;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `gets()` reads characters from the standard input stream and copies these characters to the string pointed to by `s`, until a newline character or an end-of-file marker is encountered. Any newline character is discarded, and a null character is appended to the string.

Return values: ... `gets()` returns `s` if reading was successful. If an end-of-file marker is encountered or if an error occurs while reading the `NULL` pointer is returned.

Notes and remarks: If `gets()` was unsuccessful (i.e. as an effect of `MAX_LINE`, see `stdio.h`), the contents of the character array pointed to by `s` depend on the number of characters read into the array so far. This implies that the resulting array may not be terminated by a null character, and therefore cannot be treated as a string.

See also: .... `fgets()`

`stdio.h`

GetSysInfo()

Usage: ....
```
#include <system.h>
int GetSysInfo(synfdest);
SYSTEM *synfdest;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `GetSysInfo()` gets information concerning the transputer where this function is called and assigns this information to the structure pointed to by `synfdest`.

The information can be used to direct the flow of control related to message passing, load balancing, inter processor communication etc.

Return values:.... `GetSysInfo()` returns zero on success and nonzero when no information is available.

See also: .... `GetNodeInfo()`

Example : .... In the `\examples` directory a number of programs make use of this routine to obtain local system information, e.g. for message passing and network information display. Here is a short example on how to call it.

```
#include system.h
main()
{
   SYSTEM sys;
   GetSysInfo(&sys);
}
```

gmtime()

Usage: ....

```
#include <time.h>
struct tm *gmtime(timer);
time_t *timer;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: .... `gmtime()` converts the encoded value of the calendar time pointed to by `timer` to a broken-down time representing Greenwich Mean Time and sets the members of the static `tm` structure to the appropriate values.

Return values: ... `gmtime()` returns a pointer to the `tm` structure in the static data area which is set to indicate Greenwich Mean Time, which is calculated using the encoded value of the `time_t` pointed to by `timer`.

Notes and remarks: .... Since this and related time and date functions have been defined to operate on a structure and a character array in the static data area, calls to other functions may overwrite the result of a call to `gmtime()`. Also, other processes calling the same routine will cause the result to be overwritten. To avoid these effects all non re-entrant time and date functions have been given a re-entrant replacement in the Par.C System. The replacement for `gmtime()` is `gmtime_r()`.

See also: ....

`gmtime_r()`

`time()`

`time.h`

gmtime_r()

Usage: ....

```
#include <time.h>
struct tm *gmtime_r(timer, ptr);
time_t *timer;
struct tm *ptr;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `gmtime_r()` converts the encoded value of the calendar time pointed to by `timer` to a broken-down time representing Greenwich Mean Time and sets the members of the `tm` structure pointed to by `ptr` to the appropriate values.

This function is the re-entrant replacement for the Draft Proposed ANSI C function `gmtime()`. Instead of writing the result of the conversion into a `tm` structure in the static data area, a pointer to the resulting structure is passed to the function from the caller.

Return values:.... `gmtime_r()` returns a pointer to the `tm` structure containing the broken-down time representing the current Greenwich Mean Time, using the encoded value of the `time_t` pointed to by `timer`.

See also: .... `gmtime()`

`time()`

`time.h`

_in()

Usage: ....
```
#include <transp.h>
void _in(src, dest, size)
channel *src;
char *dest;
int size;
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: .... The function `_in()` receives `size` bytes from the channel pointed to by `src` in the buffer pointed to by `dest`.

Return values: ... none

See also: .... `_out()`

Example: ....
```
#include <stdio.h>
#include <string.h>
#include <transp.h>

void main()
{
   channel comm;
   par
   {{
       static char src[] = "This is the source string.\n";
       printf("(sending process) %s", src);
       comm = strlen(src)+1;     /* send length of string */
       _out(&comm,src,strlen(src)+1);    /* send contents */
   }{
       static char dst[] =
                       "This is the destination string.\n";
       printf("(receiving process) %s", dst);
       _in(&comm,dst,(int)comm); /* receive other string */
       printf("(receiving process) %s", dst);
   }}
}
```

InMess()

Usage: ....
```
#include <system.h>
void   InMess(message, size, from);
char   *message;
int size;
channel *from;
```

Availability: ...... Par.C System

Library: .... `a.lib` and `b.lib`

Description: ..... `InMess()` reads size bytes from the channel pointed to by `from` into the character array pointed to by `message`.

Return values: .... none

Notes and remarks: This is the C-implementation of the transputer instruction "in". The only difference is that no communication is performed if size is zero. This implies that at least one byte must be transferred, even when only synchronisation is desired. The proper working of `InMess()` is only guaranteed, if exactly one other process sends data of the same size over the same channel.

See also: .... `OutMess()`

`RecvLink()`

`RecvLinkOrFail()`

`SendLink()`

`SendLinkOrFail()`

`_in()`

`_out()`

InPort()

Usage: ....

```
#include <stdlib.h>
void InPort(prtaddr, size, buff);
int prtaddr, size;
char *buff;
```

Availability: ..... Par.C System

Library: .... `io.lib`

Description: ..... `InPort()` reads `size` bytes from host I/O port with address `prtaddr`. The bytes are stored in a buffer located at the transputer with startaddress `buff`.

This function is intended for hosts that have CPU's with separate instructions for I/O, controlling a dedicated I/O bus. What ports are accessible depends on the operating system on the host.

Return values: ... none

Notes and remarks: Care should be taken with forming `prtaddr`, which should be recognisable by the host as a valid port address.

See also: ....

`OutPort()`

`PeekHost()`

`PokeHost()`

IntToDate()

Usage: ....
```
#include <time.h>
int    IntToDate(lindat, pday, pmonth, pyear);
int    lindat, *pday, *pmonth, *pyear;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `IntToDate()` converts an integer assumed to contain a linear date, to the corresponding day, month and year. Month ranges from 0-11, the year includes the right century.

Notes & Remarks: 0 is converted to day 1, month 2 (March) and year 1972.

After conversion the values of day, month and year are stored in locations pointed to by respectively `pday`, `pmonth` and `pyear`. Dividing a value of type `time_t` as returned by the routines `mktime()` and `time()` by 86400 (number of seconds in a day), will yield a linear date, which is convertible by `IntToDate()`.

Return values:.... `IntToDate()` returns zero.

See also: .... `DateToInt()`

isalnum()

Usage: ....
```
#include <ctype.h>
int isalnum(c);
char c;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: .... Routine to test whether the character `c` is in the range of alphanumeric characters.

Return values: ... `isalnum()` returns a nonzero value if value for a character in one of the ranges 0 to 9, a to z, or A to Z. Otherwise, the returned value is zero.

See also: ....

`isalpha()`

`isdigit()`

`isascii()`

isalpha()

Usage: ....
```
#include <ctype.h>
int isalpha(c);
char c;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... Routine to test whether the character `c` is in the range of upper or lower case letters.

Return values:.... `isalpha()` returns a nonzero value if `c` is the ASCII code value for a character in one of the ranges a to z or A to Z. Otherwise, the returned value is zero.

See also: .... `isalnum()`

`isascii()`

isascii()

Usage: ....
```
#include <ctype.h>
int isascii(c);
char c;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... Routine to test whether the character `c` is in the range of ASCII coded characters.

Return values:.... `isascii()` returns a nonzero value if the value of `c` is in the range of ASCII codes (values 0 through 0x7F). Otherwise, the returned value is zero. A table of ASCII coded characters is included in the appendices.

See also: .... `toascii()`

iscntrl()

Usage: ....
```
#include <ctype.h>
int iscntrl(c);
char c;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: .... Routine to test whether the character `c` is in the range of control characters.

Return values: ... `iscntrl()` returns a nonzero value if the value of `c` is in the range of control codes, i.e. any character that is not a printable character. Otherwise, the returned value is zero. If the ASCII character set is used, the control characters have values 0 through 0x1F or value 0x7F. A list of ASCII coded control characters is given in the appendices.

See also: .... `isprint()`

isdigit()

Usage: ....
```
#include <ctype.h>
int isdigit(c);
char c;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: .... Routine to test whether the character `c` is one of the decimal digits.

Return values: ... `isdigit()` returns a nonzero value if the value of `c` is the code for one of the decimal digits 0 through 9. Otherwise, the returned value is zero.

See also: .... `isxdigit()`

`isodigit()`

isgraph()

Usage: ....
```
#include <ctype.h>
int isgraph(c);
char c;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... Routine to test whether the character `c` is one of the printable characters, except for the space character.

Return values:.... `isgraph()` returns a nonzero value if the value of `c` is the code for one of the printable characters, except for the space character. Otherwise, the returned value is zero.

See also: .... `isprint()`

`isspace()`

`iswhite()`

islower()

Usage: ....

```
#include <ctype.h>
int islower(c);
char c;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: .... Routine to test whether the character `c` is a lower case letter.

Return values: ... `islower()` returns a nonzero value if the value of `c` is the code for one of the lower case letters a through z. Otherwise, the returned value is zero.

See also: .... `isalpha()`

`isupper()`

`tolower()`

isodigit()

Usage: ....

```
#include <ctype.h>
int isodigit(c);
char c;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... Routine to test whether the character `c` is one of the octal digits.

Return values: .... `isodigit()` returns a nonzero value if the value of `c` is the code for one of the octal digits 0 through 7. Otherwise, the returned value is zero.

See also: .... `isdigit()`

`isxdigit()`

isprint()

Usage: ....
```
#include <ctype.h>
int isprint(c);
char c;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: .... Routine to test whether the character `c` is one of the printing characters, i.e. not a control character.

Return values: ... `isprint()` returns a nonzero value if the value of `c` is the code for one of the printable characters, i.e. not a control character. Otherwise, the returned value is zero. If the ASCII character set is used, the printable characters have codes 0x20 through 0x7E.

See also: .... `iscntrl()`

`isascii()`

ispunct()

Usage: ....
```
#include <ctype.h>
int ispunct(c);
char c;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... Routine to test whether the character `c` is one of the punctuation characters, i.e. not a control character or an alphanumeric character.

Return values:.... `ispunct()` returns a nonzero value if the value of `c` is the code for one of the punctuation characters, i.e. not a control character or an alphanumeric character. Otherwise, the returned value is zero.

See also: .... `iscntrl()`

`isalnum()`

isspace()

Usage: ....

```
#include <ctype.h>
int isspace(c);
char c;
```

Availability: ..... Draft Proposed ANSI C

Library:......... `std.lib`

Description: .... Routine to test whether the character `c` is a whitespace character. In the implementation of this function, the meaning of whitespace is taken in the broad sense, which implies that the following characters fall in the set of whitespace characters:

'\t'(0x9) : horizontal tab	'\n' (0xA) : newline
'\f' (0xB) : form feed	'\v' (0xC) : vertical tab
'\r' (0xD) : carriage return	' ' (0x20) : blank or space

Return values: ... `isspace()` returns a nonzero value if the value of `c` is the code for a whitespace character in the broad sense, i.e. a horizontal tab, newline, form feed, vertical tab, carriage return or a space character. Otherwise, the returned value is zero.

See also: .... `iswhite()`

isupper()

Usage: ....
```
#include <ctype.h>
int isupper(c);
char c;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... Routine to test whether the character `c` is an upper case letter.

Return values:.... `isupper()` returns a nonzero value if the value of `c` is the code for one of the upper case letters A through Z. Otherwise, the returned value is zero.

See also: .... `isalpha()`

`islower()`

`toupper()`

iswhite()

Usage: ....

```
#include <ctype.h>
int iswhite(c);
char c;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: .... Routine to test whether the character `c` is a whitespace character. In this function, the meaning of whitespace is taken in a restricted sense, which implies that the following characters fall in the set of whitespace characters:

'\t' (0x9) : horizontal tab ' ' (0x20) : blank or space

Return values: ... `iswhite()` returns a nonzero value if the value of `c` is the code for a whitespace character in the restricted sense, i.e. a horizontal tab or a space character. Otherwise, the returned value is zero.

See also: .... `isspace()`

isxdigit()

Usage: ....
```
#include <ctype.h>
int isxdigit(c);
char c;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... Routine to test whether the character `c` is one of the hexadecimal digits.

Return values:.... `isxdigit()` returns a nonzero value if the value of `c` is the code for one of the hexadecimal digits, i.e. if the coded character is in one of the ranges 0 through 9, a through f or A through F. Otherwise, the returned value is zero.

See also: .... `isdigit()`

`isodigit()`

itoa()

Usage: ....

```
#include <stdcnv.h>
int itoa(num, str)
int num;
char *str;
```

Availability: ..... Par.C System

Description: ..... The macro `itoa()` converts an integer to its ASCII representation in decimal notation. Both the integer and a pointer to the string are passed as arguments. The string should be sufficiently long to store the result of the conversion.

Return values: .. The length of the converted string.

See also: .... `_toa()`

labs()

Usage: . . . .
```
#include <stdlib.h>
long int labs(x);
long int x;
```

Availability: . . . . . . Draft Proposed ANSI C

Library: . . . . `std.lib`

Description: . . . . . `labs()` computes the absolute value of `x`, which is expected to be a (signed) long integer.

Return values: . . . . `labs()` returns the absolute value of `x`.

See also: . . . . `abs()`
`fabs()`

ldexp()

Usage: ....

```
#include <math.h>
double ldexp(x, exp);
double x;
int exp;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `math.lib` and `t8math.lib`

Description: ..... `ldexp()` multiplies a floating-point number by an integral power of 2. The argument `x` is expected to have the type double, and the second argument `exp` is taken as an integer value.

Return values: ... `ldexp()` returns the value of $x * 2^{exp}$.

leave()

This function has been removed from the Par.C System runtime libraries. The necessary support in the Par.C loader/server could not be implemented in a correct way, since the functioning of `leave()` would allow the loader/server to exit without closing files and flushing buffers.

The possibility of communicating in a more direct way with the host system is implemented through the use of `PeekHost()` and `PokeHost()`. Doing communications with a user-defined server can be done through the `system()` function, or by calling `Release()`. This last option implies that the file I/O cannot be re-installed again. If this possibility is needed, we can supply you with the necessary information on demand.

See also: .... `PeekHost()`
`PokeHost()`
`system()`
`Release()`

LINKIN()

Usage: ....

```
#include <system.h>
char * LINKIN(Linkno);
int Linkno;
```

Availability: ..... Par.C System

Description: ..... The macro `LINKIN()` evaluates to the address of the incoming channel of the link `Linkno`. The resulting address can be used to implement direct channel communication to another transputer. However, to remain compatible with future versions of the Par.C System, it is advised to use the functions `RecvLink()` or `RecvLinkOrFail()` instead.

Return values: ... `LINKIN()` returns a pointer to the incoming channel of link `Linkno`.

Notes and remarks: This macro is not garanteed to be compatible with future versions of the Par.C System.

See also: ....

`RecvLink()`
`RecvLinkOrFail()`
`LINKOUT()`

LINKOUT()

Usage: . . . .
```
#include <system.h>
char * LINKOUT(Linkno);
int Linkno;
```

Availability: . . . . . . Par.C System

Description: . . . . . The macro `LINKOUT()` evaluates to the address of the outgoing channel of the link `Linkno`. The resulting address can be used to implement direct channel communication to another transputer. However, to remain compatible with future versions of the Par.C System, it is advised to use the functions `sendLink()` or `sendLinkorFail()` instead.

Return values: . . . . `LINKOUT()` returns a pointer to the outgoing channel of link `Linkno`.

Notes and remarks: This macro is not garanteed to be compatible with future versions of the Par.C System.

See also: . . . .
```
sendLink()
sendLinkOrFail()
LINKIN()
```

Example: . . . .
```
#include <system.h>
int main()
{
   int varia = 0xabcd;
   if(_Tn == 1){
       ...
       *LINKOUT(3) = varia;     /* send over link three */
       ...
   }else{
       ...
       varia = *LINKIN(1);      /* receive from link one */
       ...
   }
}
```

localtime()

Usage: ....

```
#include <time.h>
struct tm *localtime(timer);
time_t *timer;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `localtime()` converts the encoded value of the calendar time pointed to by `timer` to a broken-down time representing the local time and sets the members of the static `tm` structure to the appropriate values.

Return values: ... `localtime()` returns a pointer to the `tm` structure in the static data area which is set to indicate the local time, which is calculated using the encoded value of the `time_t` pointed to by `timer`.

Notes and remarks: .... Since this and related time and date functions have been defined to operate on a structure and a character array in the static data area, calls to other functions may overwrite the result of a call to `localtime()`. Also, other processes calling the same routine will cause the result to be overwritten. To avoid these effects the most important non re-entrant time and date functions have been given a re-entrant replacement in the Par.C System. The replacement for `localtime()` is `localtime_r()`.

See also: ....

```
localtime_r()
time()
time.h
```

localtime_r()

Usage: ....
```
#include <time.h>
struct tm *localtime_r(timer, ptr);
time_t *timer;
struct tm *ptr;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `localtime_r()` converts the encoded value of the calendar time `timer` to a broken-down time representing the local time, and sets the members of the `tm` structure pointed to by `ptr` to the appropriate values.

This function is the re-entrant replacement for the Draft Proposed ANSI C `localtime()`. Instead of writing the result of the conversion into a `tm` structure in the static data area, a pointer to the resulting structure is passed to the function from the caller.

Return values: ... `localtime_r()` returns a pointer to the `tm` structure containing the broken-down time representing the local time, using the encoded value of the `time_t` given in `timer`.

See also: ....
`localtime()`
`time()`
`time.h`

log()

Usage: ....

```
#include <math.h>
double log(x);
double x;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `math.lib` and `t8math.lib`

Description: ..... `log()` computes the natural logarithm of `x`. The argument `x` is expected to have the type double.

Return values: ... `log()` returns the value of the natural logarithm of `x`.

log10()

Usage: . . . .
```
#include <math.h>
double log10(x);
double x;
```

Availability: . . . . . . Draft Proposed ANSI C

Library: . . . . `math.lib` and `t8math.lib`

Description: . . . . . `log10()` computes the base-ten logarithm of `x`. The argument `x` is expected to have the type double.

Return values: . . . . `log10()` returns the value of the base-ten logarithm of `x`.

ltoa()

Usage: ....
```
#include <stdcnv.h>
int ltoa(num, str)
long int num;
char *str;
```

Availability: ..... Par.C System

Description: ..... The macro `ltoa()` converts a long integer (64 bits) to its ASCII representation in decimal notation. Both the integer and a pointer to the string are passed as arguments. The string should be sufficiently long to store the result of the conversion.

Return values: .. The length of the converted string.

See also: .... `_toa()`

malloc()

Usage: ....

```
#include <stdlib.h>
char *malloc(size);
size_t size;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `malloc()` causes a memory block of `size` bytes to be allocated and a pointer to the allocated block to be returned. If no block of at least `size` bytes could be allocated, the `NULL` pointer is returned.

Return values: .... `malloc()` returns a pointer to the data area of an allocated block of at least `size` bytes, if allocation was successful. Otherwise, the `NULL` pointer is returned.

Notes and remarks: The method of allocation used by `malloc()` is optimised for size, which implies that only `size` bytes + some words overhead are allocated. All blocks are aligned to word boundaries and the value returned is a pointer to the data area of the allocated block.

`malloc()` allocates blocks of memory by a "first fit" method, and starts the search for available blocks from the far end of memory. The corresponding function `smalloc()` can be used to obtain fast on-chip memory from the first 2 or 4 KBytes which are available in the transputer.

High priority processes performing a `malloc()` will temporarily be switched to low priority while allocating memory.

See also: ....

```
smalloc()
calloc()
free()
realloc()
MemAvail()
```

matherr()

Usage: ....

```
#include <math.h>
double matherr(code,arglist)
double _matherr(code,arglist)
int code;
_MATHARGS arglist;
```

Availability: ..... Par.C System

Library: .... `math.lib` and `t8math.lib`

Description: ..... The math functions in `math.lib` and `t8math.lib` call the function `matherr()` when an error occurs. They pass to it an integer code, indicating which error occured, and a pointer to the parameter list. The error codes are contained in `errno.h`.

The function `matherr()` does nothing but call `_matherr()` with the same error code and parameter pointer. Thus, the user can supply his own `matherr()` function to deal with certain kinds of errors, while calling the default error handler `_matherr()` for all other errors.

The function `_matherr()` calls the general error handler (pointed to by the function pointer `(*ERROR)()`.

Return values: ... `matherr()` returns the value returned by `_matherr()`. `_matherr()` returns a so called Not a Number (NaN), which is a 64 bit pattern with 0x7ff in the most significant 12 bits. The least significant bits contain the error code which was passed to `_matherr()`.

See also: .... `errno.h`

`(*ERROR)()`

`NaN()`

Example: ....
```
#include <errno.h>
#include <math.h>

double matherr(code,args)
int code;
_MATHARGS args;
{
       if (code==EM_POWDOMAIN && args->x.db==0.0 &&
                                    args->y.db==0.0)
           return 1.0;      /* define pow(0.0,0.0) */
       return _matherr(code,args);
                    /* default math error handler */
}
```

MemAvail()

Usage: ....

```
#include <stdlib.h>
struct MEMINFO *MemAvail(info);
struct MEMINFO *info;
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: ..... `MemAvail()` scans the complete memory of the transputer and writes information on the available free memory blocks into the `MEMINFO` structure pointed to by `info`. The `MEMINFO` structure is defined in `stdlib.h` as:

```
struct MEMINFO {
       size_t total_free;      /* total amount of free memory */
       size_t largest_free;    /* largest free block available */
       int    nr_of_blocks;    /* total number of free blocks */
   }
```

Return values: ... The value of `info`.

Notes and remarks: The information returned in the `MEMINFO` structure only gives the status at the moment `MemAvail()` was executed. Even if this call is immediately followed by a call

```
p = malloc(info->largest_free)
```

there is no guarantee that this will be successful. Another process may have allocated memory between the two calls (explicitly or implicitly through the dynamic workspace allocation).

See also: ....

```
malloc()
smalloc()
calloc()
realloc()
free()
```

memchr()

Usage: ....
```
#include <string.h>
char *memchr(s, c, n);
char *s, c;
size_t n;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `memchr()` searches for the first occurrence of the character `c` in the first `n` characters of the sequence pointed to by `s`. If the character `c` is found, a pointer to the first occurrence of `c` is returned. Otherwise the `NULL` pointer is returned.

Return values:.... `memchr()` returns a pointer to the first occurrence of `c` in the first `n` characters of the sequence `s`, or a `NULL` pointer when `c` is not found.

See also: .... `strchr()`
`strrchr()`

memcmp()

Usage: ....

```
#include <string.h>
int memcmp(obj1, obj2, n);
char *obj1, *obj2;
size_t n;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `memcmp()` takes the first `n` characters of the objects pointed to by `obj1` and `obj2` and compares them. If the values of these first `n` characters of both objects are equal, zero is returned. Otherwise, a non-zero value is returned, according to the compared values of `obj1` and `obj2`.

Return values: ... `memcmp()` returns zero if the first `n` characters of objects `obj1` and `obj2` are identical. A non-zero value is returned if the character sequences are not identical: negative when the value of the indicated character sequence of `obj1` is smaller, positive if it is greater than the value of the corresponding character sequence of `obj2`.

See also: .... `strcmp()`
`strncmp()`

memcpy()

Usage: ....

```
#include <string.h>
char *memcpy(dest, src, n);
char *dest, *src;
size_t n;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `memcpy()` copies the `n` characters from the object pointed to by `src` to the first `n` characters of the object pointed to by `dest`. The result is not defined in case of overlapping source and destination objects.

Return values:.... `memcpy()` returns the value of `dest`.

See also: .... `strcpy()`
`memmove()`

memfill()

Usage: ....

```
#include <string.h>
char *memfill(obj, ptr, size, cnt);
char *obj, *ptr;
int size, cnt;
```

Availability: ..... Par.C System

Library:......... `std.lib`

Description: ..... `memfill()` fills the object pointed to by `obj` with the pattern of `size` bytes pointed to by `ptr`, which is repeated `cnt` times.

Return values: .. `memfill()` returns `obj`

Notes and remarks: The number of bytes written is `cnt * size`. No 'stride' can be defined, i.e. each copy of the pattern follows the former immediately.

See also: .... `memset()`

Example: ....

```
#include <string.h>

...

   channel *MultiChan;
   int ResetPat = MostNeg; /* predefined as 0x80000000 */
   int NrOfChannels = 1024; /* number of channels desired */

/* use memfill to reset dynamically allocated channels */

   MultiChan = (channel *) malloc( NrOfChannels *
       sizeof( channel ) );
   memfill( MultiChan, &ResetPat, sizeof( channel ),
       NrOfChannels );
```

memmove()

Usage: ....
```
#include <string.h>
char *memmove(dest, src, n);
char *dest, *src;
size_t n;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `memmove()` copies the `n` characters from the object pointed to by `src` to the first `n` characters of the object pointed to by `dest`.

Return values:.... `memmove()` returns the value of `dest`.

Notes and remarks: In contrast with `memcpy()`, `memmove()` renders correct results, even if the source and destination overlap.

See also: .... `memcpy()`
`strcpy()`

memset()

Usage: ....
```
#include <string.h>
char *memset(obj, c, n);
char *obj;
int c;
size_t n;
```

Availability: ..... Draft Proposed ANSI C

Library:......... `std.lib`

Description: ..... `memset()` copies the value of `c`, converted to an unsigned char type, into each of the first `n` characters of the object pointed to by `obj`.

Return values: ... `memset()` returns `obj`.

See also: .... `memfill()`

mktime()

Usage: ....

```
#include <time.h>
time_t mktime(ptr);
struct tm *ptr;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `mktime()` converts the time information contained in the `tm` structure pointed to by `ptr` to a calendar time value of type `time_t`, which has the same encoding as the values returned by `time()`. The original values of the `tm_wday` and `tm_yday` elements of the indicated structure are ignored, and the values of the other elements are not restricted to the ranges which they would have when obtaining the structure from `gmtime()` or `localtime()`. When execution of `mktime()` was successful, the elements of the `tm` structure are set to the correct values and the returned value is a valid representation of the indicated calendar time represented as `time_t`.

Return values:.... `mktime()` returns the calendar time encoded in a `time_t` type. If the time indicated in the `tm` structure pointed to by `ptr` cannot be represented, the value `(time_t)(-1)` is returned.

See also: ....

`time()`
`gmtime()`
`localtime()`
`time.h`

modf()

Usage: ....

```
#include <math.h>
double modf(x, iptr);
double x, *iptr;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `math.lib` and `t8math.lib`

Description: ..... `modf()` breaks the floating-point number `x` into integral and fractional parts, each of which has the same sign as the argument. The argument `x` is expected to have the type double. The computed integral part is stored as a double value in the object pointed to by `iptr`.

Return values: ... `modf()` returns the fractional part of `x`.

NaN()

Usage: ....
```
#include <math.h>
int NaN (x);
double x;
```

Availability: ...... Par.C System

Library: .... `math.lib` and `t8math.lib`

Description: ..... The function `NaN()` checks whether its argument `x` is a Not a Number.

Return values: .... The function `NaN()` returns 0 if `x` is a regular double. If `x` is a NaN (Not a Number) generated by `_matherr()`, it will return the associated error code, as defined in `errno.h`. If `x` is a NaN not generated by `_matherr()`, 1 is returned.

See also: ....
`matherr()`
`_matherr()`

Example: ....
```
double x,y,z;
...
x=y*z;
if (NaN(x)){
        printf("Exception occurred.\n");
        exit(1);
}
```

offsetof()

Usage: ....

```
#include <stddef.h>
size_t offsetof(s_name, m_name);
structure name s_name;
member name    m_name;
```

Availability: ..... Draft Proposed ANSI C

Library: .... defined as a macro in `stddef.h`

Description: ..... `offsetof()` is defined as macro in `stddef.h` and is used to obtain the offset in bytes of a structure member identified by `m_name` in the structure type designated by `s_name`. The type of the value replacing the macro is `size_t`.

See also: .... `stddef.h`

oltoa()

Usage: ....
```
#include <stdcnv.h>
int oltoa(num, str)
long int num;
char *str;
```

Availability: ...... Par.C System

Description: ..... The macro `oltoa()` converts a long integer (64 bits) to its ASCII representation in octal notation. Both the integer and a pointer to the string are passed as arguments. The string should be sufficiently long to store the result of the conversion.

Return values: ... The length of the converted string.

See also: .... `_toa()`

onexit()

Usage: ....
```
#include <stdlib.h>
int onexit(fptr);
void (*fptr)();
```

Availability: ..... Par.C System

Library:......... `std.lib`

Description: ..... See `atexit()`.

Notes and remarks: `onexit()` is defined as a macro:
```
#define onexit(func) atexit(func)
```

See also: .... `atexit()`

otoa()

Usage: ....
```
#include <stdcnv.h>
int otoa(num, str)
int num;
char *str;
```

Availability: ...... Par.C System

Description: ..... The macro `otoa()` converts an integer to its ASCII representation in octal notation. Both the integer and a pointer to the string are passed as arguments. The string should be sufficiently long to store the result of the conversion.

Return values:.... The length of the converted string.

See also: .... `_toa()`

_out()

Usage: ....

```
#include <transp.h>
void _out(dest, src, size)
channel *dest;
char *src;
int size;
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: ..... The function `_out()` sends `size` bytes to the channel pointed to by `dest` from the buffer pointed to by `src`.

Return values: ... none

See also: .... `_in()`
`_SendLink()`

OutMess()

Usage: ....

```
#include <system.h>
void OutMess(message, size, to);
char *message;
int size;
channel *to;
```

Availability: ...... Par.C System

Library: .... `a.lib` and `b.lib`

Description: ..... `OutMess()` sends `size` bytes over the channel pointed to by `to` from the character array pointed to by `message`.

Return values:.... none

Notes and remarks: This is the C-implementation of the transputer instruction `out`. The only difference is that no communication is performed if `size` is zero. This implies that at least one byte must be transferred, even when only synchronisation is desired. The proper working of `OutMess()` is only guaranteed, if exactly one other process expects data of the same size over the same channel.

See also: .... `InMess()`
`SendLink()`
`SendLinkOrFail()`

OutPort()

Usage: ....

```
#include <stdlib.h>
void OutPort(prtaddr, size, buff);
int prtaddr, size;
char *buff;
```

Availability: ..... Par.C System

Library: .... `io.lib`

Description: ..... `OutPort()` writes `size` bytes to a host I/O port with address `prtaddr`. The bytes are fetched from a buffer located at the transputer with startaddress `buff`.

This function is intended for hosts that have CPU's with separate instructions for I/O, controlling an I/O dedicated bus. What ports are accessible is host dependent.

Return values: ... none

Notes and remarks: IMPORTANT NOTE: GREAT CARE SHOULD BE TAKEN WHEN USING THIS FUNCTION.

Care should be taken with forming `prtaddr` which, should be recognizable by the host as a valid port address.

Also byte and bit ordering may differ on the host. The bytes and bits should then be rearranged to match the ordering used on the host.

See also: ....

`InPort()`
`PeekHost()`
`PokeHost()`

P()

Usage: ....
```
#include <stdlib.h>
void P(SemPtr);
semaphore *SemPtr;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `P()` implements semaphores, together with the `V()` function and the semaphore type definition. `P()` operates on a user-declared semaphore and causes the processes accessing the same semaphore to be queued in a First In First Out queue.

Semaphores can be used by processes to share resources or variables. In the Par.C System, semaphores are integer pointers and are used to implement a queue of processes waiting for the same semaphore. The queue is handled in the first-in-first-out way. Processes of different priorities are treated alike, i.e. processes running on high priority are queued at the end of the line.

Before using the `P()` and `V()` functions, a semaphore should be declared and initialised to contain the value 0.

```
semaphore MySem = 0;
P(&MySem);
...                  /* critical section of this program */
V(&MySem);
```

When a process calls `P()` with `MySem` having the value 0 (zero), the process is given access to its critical section and the semaphore is set to indicate the active process. The critical section is terminated when this active process calls `V()`. When another process is inside its critical section guarded by `MySem`, the process calling `P()` will be queued and temporarily stopped. The waiting process will resume execution after all preceding processes, waiting for `MySem`, have been scheduled, entered and exited their critical sections.

Notes and remarks: It is vital for correct functioning of semaphores that a critical section in a process is started with a `P()` operation and ended with a `V()` operation. Performing a `V()` operation without a corresponding previous `P()` operation will void the protection of the resource or variable. This may cause loss of active processes.

Forgetting to call `V()` will cause all following code to be inside the critical section: other processes will never obtain access to the shared resources or variables and the program will probably deadlock as a result of this.

Forgetting to initialise the semaphore to 0, will cause every process performing a `P()` operation to be descheduled forever.

Processes running in a critical section can not automatically be considered as non-interruptable. The process is scheduled and de-scheduled as usual. However, the processes waiting for the semaphore will not interrupt this process before it releases it.

Both high and low priority processes can share one semaphore, but high priority processes are switched to low priority temporarily while obtaining it.

Example: ....

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

#define N 10

semaphore ComSemaphore = 0;
channel ComChannel;

void main()
{
   par
   {{
   /* concurrently send N messages of different length */
       int i,j;
       par(i=1; i<= N; i++)
       {{
           P(&ComSemaphore);    /* Claim channel and wait */
                                /* when it is occupied */
           for(j=1; j<=i; j++) /* Send some integers */
           {
               ComChannel = j;
           }
           ComChannel = 0; /* Send an "end of message" */
           V(&ComSemaphore); /* Release the channel */
       }}
       ComChannel = -1; /* Send an "end of all messages" */
   }{
          /* Receives all messages and print the values sent */
              int j;
              j = ComChannel;
              while( j != -1 ) /* Stop at "end of all messages" */
              {
                  printf("%d  ",j);
                  if(j==0) printf("\n");
                               /* new line at "end of message" */
                  j = ComChannel;
              }
          }}
      }
```

PeekHost()

Usage: ....

```
#include <stdlib.h>
void PeekHost(memaddr, size, buff);
int memaddr, size;
char *buff;
```

Availability: ..... Par.C System

Library: .... `io.lib`

Description: ..... `PeekHost()` reads `size` bytes of consecutive memory at the host, starting at `memaddr`. The bytes are put in a buffer located at the transputer pointed to by `buff`.

This function is intended to provide direct access to the memory of the host. What memory is accessible is host dependent.

Return values: ... none

Notes and remarks: Care should be taken with forming `memaddr` which, should be recognizable by the host as a valid memory address.

Byte and bit ordering may differ on the host. The bytes and bits should then be rearranged.

See also: ....

`InPort()`
`OutPort()`
`PokeHost()`

perror()

Usage: ....

```
#include <stdio.h>
void perror(lmessage);
char *lmessage;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `perror()` prints a message to `stderr` (standard error stream). First the user-supplied leader message `lmessage`, then a colon and space, then the system error message, then a new-line character.

The system error message is a textual mapping of the `errno` variable, the values of `errno` and related text can be found in the include file `errno.h`. Before calling a library routine `errno` should be cleared, and `perror()` must be called immediately after the call to the library routine (see notes).

The `errno` variable should be declared at an external level by means of including `<stddef.h>`.

The argument `lmessage` may be a `NULL` pointer or point to a null string, which results in an empty leader message.

Return values:.... none

Notes and remarks: In case parallel processes call library routines that can set the global `errno` variable, one cannot rely on the value of `errno` and hence neither on the routines that use `errno`. In case the `errno` variable contains a value wich cannot be mapped, the system error message "error not found" is printed.

See also: ....

`errno.h`
`stddef.h`
`strerror()`
`(*ERROR)()`

PokeHost()

Usage: ....

```
#include <stdlib.h>
void PokeHost(memaddr, size, buff);
int memaddr, size;
char *buff;
```

Availability: ..... Par.C System

Library: .... `io.lib`

Description: ..... `PokeHost()` writes `size` bytes to consecutive memory addresses at the host, starting at `memaddr`. The bytes are fetched from a buffer located at the transputer at address `buff`.

This function is intended to provide direct access to the memory of the host. The memory that is accessible is host dependent.

Return values: ... none

Notes and remarks: Care should be taken with forming `memaddr`, which should be recognizable by the host as a valid memory address.

Byte and bit ordering may differ on the host. The bytes and bits should then be rearranged.

See also: ....

`InPort()`
`OutPort()`
`PeekHost()`

pow()

Usage: ....
```
#include <math.h>
double pow(x, y);
double x, y;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `math.lib` and `t8math.lib`

Description: ..... `pow()` computes the value of `x` raised to the power `y`. The arguments `x` and `y` are expected to have the type double. An error occurs when `x` is zero and `y` is less than or equal to zero, or if `x` is negative and `y` is not an integer. Also, a range error may occur if the return value cannot be represented in a double.

Return values:.... `pow()` returns the value x^y.

See also: .... `errno.h`

pow2()

Usage: ....

```
#include <math.h>
double pow2(x);
double x;
```

Availability: ..... Par.C System

Library: .... `math.lib` and `t8math.lib`

Description: ..... `pow2()` computes the value of 2 raised to the power `x`. The argument `x` is expected to have the type double. A range error may occur if the return value cannot be represented in a double.

Return values: ... `pow2()` returns the value 2^x.

See also: .... `errno.h`

pow10()

Usage: ....
```
#include <math.h>
double pow10(x);
double x;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `math.lib` and `t8math.lib`

Description: ..... `pow10()` computes the value of 10 raised to the power `x`. The argument `x` is expected to have the type double. A range error may occur if the return value cannot be represented in a double.

Return values:.... `pow10()` returns the value 10^{x}.

See also: .... `errno.h`

printf()

Usage: ....

```
#include <stdio.h>
int printf(format, ...);
char *format;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `printf()` writes output to the standard output stream in the format indicated in the format string pointed to by `format`, using the optional arguments following the format string.

The format string may contain zero or more ordinary characters which are copied unchanged to the standard output stream and zero or more conversion specifications, resulting in one or more subsequent arguments being converted to printing format and written to the standard output stream. Each conversion specification is preceded by the character %. Writing the character % to the standard output stream is achieved by putting %% in the format string.

A full description of the possible conversion specifications and flags is given in the description of the related `fprintf()`. The difference between `fprintf()` and `printf()` is, that the latter writes to the standard output stream rather than to a stream indicated by the first argument.

Return values: ... `printf()` returns the number of characters written to the standard output stream, or a negative value if an error occurred.

See also: ....

```
fprintf()
sprintf()
vprintf()
vsprintf()
vfprintf()
```

Priority()

Usage: .... `#include <system.h>`
`int Priority();`

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `Priority()` obtains the priority of the current process.

Return values:.... `Priority()` returns the value of one of the defined constants `HIGH_PRIORITY` or `LOW_PRIORITY` dependending on the current priority.

See also: .... `SetPriority()`

putc()

Usage: ....
```
#include <stdio.h>
int putc(c, stream);
int c;
FILE *stream;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `putc()` writes the character specified by `c` and converted to an unsigned integer, to the stream pointed to by `stream`..

Return values: .... `putc()` returns a positive integer representing the character written to the stream. If an error occurs while writing, the error indicator is set and `EOF` is returned.

Notes and remarks: `putc()` is implemented as a macro, expanding to more code than a call to `fputc()` would generate, but which executes faster. Side effects in the expression `stream` should be avoided.

See also: .... `fputc()`
`putchar()`

putchar()

Usage: ....
```
#include <stdio.h>
int putchar(c);
int c;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `putchar()` writes the character specified by `c` and converted to an unsigned integer, to the standard output stream.

Return values:.... `putchar()` returns a positive integer representing the character written to the standard output stream. If an error occurs while writing, the error indicator is set and EOF is returned.

Notes and remarks: `putchar()` is implemented as a macro, expanding to `putc(c, stdout)`.

See also: .... `putc()`
`fputc()`

puts()

Usage: ....
```
#include <stdio.h>
int puts(s);
char *s;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `puts()` writes the string pointed to by `s` to the standard output stream and appends a newline character to the output. The terminating null character is not written.

Return values: ... `puts()` returns a nonzero value if the string has been successfully written to the standard output stream. Otherwise, zero is returned.

See also: .... `fputs()`

qsort()

Usage: ....
```
#include <stdlib.h>
void qsort(base, n, size, compare);
char   *base;
size_t n, size;
int (*compare)(p1, p2);
```

Availability: ...... Draft proposed ANSI C

Library: .... `std.lib`

Description: ..... `qsort()` takes `base` as a pointer to the array to be sorted, `n` as the number of elements in the array, `size` as the size of an element and `compare` as a pointer to the comparison function to be used. See the description of `bsearch()` for more details on `compare()`.

Return values: .... none

Notes and remarks: `qsort()` is an implementation of the quicksort algorithm. It is written recursively, so it needs much stack if the input array is large.

See also: .... `bsearch()`

raise()

Usage: ....

```
#include <signal.h>
int raise(sig);
int sig;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `raise()` provides, together with `signal()` and a number of defined signal macro's, a primitive exception handling mechanism to be used to influence the normal flow of the C program. When `raise()` is called, the routine corresponding to `sig`, as installed with `signal()`, will be executed. The function will be replaced by `SIG_DFL` in the trap table, and must be re-installed when it should be called again when `sig` is raised the next time. This can be done in the signal handling routine itself, by calling `signal()`.

Return values: ... `raise()` returns zero if it was successful, nonzero if unsuccessful.

Notes and remarks: Certain signals may be raised without explicitly calling `raise()`. This is true for `ABORT_SIG`, which is raised by `abort()` or ^C (ctrl C), for `EVENT_SIG`, which is raised by asserting the transputer's event pin, and for `ALARM_SIG`, which is raised by the alarm timer.

Raising `ALARM_SIG` will not cause sleeping processes to be woken, and raising `ABORT_SIG` will not cause the program to be aborted. `raise()` calls a function associated to `sig` and causes it to be removed from the trap table.

See also: ....

```
signal()
abort()
alarm()
```

rand()

Usage: .... `#include <stdlib.h>`
`int rand();`

Availability: ...... Draft proposed ANSI C

Library: .... `std.lib`

Description: ..... `rand()` generates a pseudo random number in the range 0 - `RAND_MAX`. `RAND_MAX` is defined in `stdlib.h`.

Return values:.... `rand()` returns the generated random number.

Notes and remarks: `rand()` is implemented as a linear feedback shift register with a 55 element buffer. The period is at least 2^{55}.

`srand()` can be used to initiate a reproducible sequence. However, when `rand()` is called from a number of concurrent processes, this reproducibility can not be guaranteed, because the sequence will be shared by these processes.

`rand()` will generate the same sequence on all transputers in the network if no transputer specific initialization is performed. `srand()` and the network info in the `SYSTEM` structure can be used to render independant results on all transputers. See the `system.h` include file for a description of the `SYSTEM` structure.

See also: .... `srand()`
`stdlib.h`
`system.h`

Example: .... A function returning a double value in the range [0.0, 1.0) can be defined as follows:

```
#include <stdlib.h>
double drand()
{
   return (double) rand() / (RAND_MAX+1.0);
}
```

realloc()

Usage: ....

```
#include <stdlib.h>
#include <stddef.h>
char *realloc(oldblock, size);
char *oldblock;
size_t size;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: .... `realloc()` attempts to adjust the size of a previously allocated block. The new block will be a storage space of `size` bytes, and will contain the information of the previously allocated block, as far as the old and new block overlap.

`oldblock` must have a value previously returned by one of the memory allocation functions. If not, `realloc()` will behave undefined. `size` is the amount of storage space to be allocated.

If necessary the contents of the old block are copied to the new block.

Return values: ... `realloc()` returns a pointer to the start (lowest byte address) of the new allocated space. A `NULL` pointer will be returned in case the space can not be allocated, or in case `size` is zero.

Notes and remarks: If argument `oldblock` is the `NULL` pointer, `realloc()` behaves like `malloc()`. The return value may differ from `oldblock`.

See also: ....

```
malloc()
smalloc()
free()
calloc()
```

RecvLink()

Usage: ....
```
#include <system.h>
int RecvLink(LinkNo, MessPtr, MessSize);
int LinkNo;
char *MessPtr;
int MessSize;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `RecvLink()` will attempt to receive a message of `Messsize` bytes from the input link indexed by `LinkNo` and write the data into the array pointed to by `MessPtr`. Error codes are generated when the function is called with an illegal `LinkNo` or when the status of the link (as given in the `SYSTEM` structure) indicates I/O errors or the link not being connected.

Return values:.... `RecvLink()` returns zero if the specified number of bytes were successfully received. If the indexed link does not exist, the error code `E_LINKNO` is returned. If the linkstatus indicates a non-connected link or a link showing I/O errors, the error code `E_CANTRECV` is returned.

Notes and remarks: It is possible to perform link communication in other ways, e.g. by assigning link addresses to channel pointers and using the primitives for channel communication available in the Par.C System. Although `system.h` contains the macros `LINKOUT` and `LINKIN` for this purpose, this method of inter-processor communication is not recommended, because future expansions of the Par.C system might use the links for low-level message passing and internal system communications, or for multi-processor file I/O. The functions `RecvLink()`, `RecvLinkOrFail()`, `SendLink()` and `SendLinkorFail()` are guaranteed to work, also in later versions of the Par.C System.

See also: ....
`RecvLinkOrFail()`
`SendLink()`
`SendLinkOrFail()`

RecvLinkOrFail()

Usage: ....

```
#include <stdlib.h>
#include <time.h>
int RecvLinkOrFail(LinkNo, MessPtr, MessSize, TimeOut);
int LinkNo;
char *MessPtr;
int MessSize;
clock_t TimeOut;
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: .... `RecvLinkOrFail()` will attempt to receive a message of `Messsize` bytes from the input link indexed by `LinkNo` and write the data into the array pointed to by `MessPtr`. If this is not accomplished within `Timeout` clockticks, the receive is aborted and the function will return the non-zero value `E_TIMEOUT`. Error codes are also generated when the function is called with an illegal `LinkNo` or when the status of the link indicates that I/O errors have occurred or that the link is not connected. The link status is described in the `SYSTEM` structure, which is described in `system.h`.

Return values: ... `RecvLinkOrFail()` returns zero if the specified number of bytes were successfully received within the specified time. If the indexed link does not exist, the error code `E_LINKNO` is returned. If the linkstatus indicates a non-connected link or a link showing I/O errors, the error code `E_CANTRECV` is returned. If the receive instruction was not successful within the specified time, the error code `E_TIMEOUT` is returned.

Notes and remarks: It is possible to perform link communication in other ways, e.g. by assigning link addresses to channel pointers and using the primitives for channel communication available in the Par.C System. Although `system.h` contains the macros `LINKOUT` and `LINKIN` for this purpose, this method of inter-processor communication is not recommended, because future expansions of the Par.C system might use the links for low-level message passing and internal system communications, or for multi-processor file I/O. The functions `RecvLink()`, `RecvLinkOrFail()`, `SendLink()` and `SendLinkOrFail()` are guaranteed to work, also in later versions of the Par.C System.

See also: .... `RecvLink()`
`SendLink()`
`SendLinkOrFail()`

Example: .... See `SendLinkOrFail()`

Release()

Usage: ....

```
#include <stdio.h>
void Release(status);
int status;
```

Availability: ..... Par.C System

Library: .... `io.lib`

Description: ..... `Release()` causes the file system on the transputer to be deactivated and the server running on the host system to exit with value `status`. This function can be used to have control return to the host system without aborting the running program on the transputer. On the transputer side, the link to the host system is free to the user program after `Release()` has been called. A special, user-defined server program (e.g. a graphics server) can then handle the communications.

Return values: ... none

remove()

Usage: ....
```
#include <stdio.h>
int remove(filename);
char *filename;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `remove()` causes the file with name `filename` to be removed. Subsequent attempts to open the file will fail. The effect of calling `remove()` for a file which is open depends on the operating system running on the host. An attempt which is not granted will result in an errorcode being returned.

Return values:.... `remove()` returns zero if the file was removed successfully. Otherwise, a nonzero value is returned.

rename()

Usage: ....

```
#include <stdio.h>
int rename(oldname, newname);
char *oldname, *newname;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `rename()` will cause the file with name `oldname` to receive the new name specified by `newname`. The effect of an attempt to rename a non-existing file, or to use an already existing filename as `newname`, depends on the operating system running on the host. A failing attempt to rename a file will cause an errorcode to be returned. In this case the file that was to be renamed will still be known under its old name on the host system.

Return values: ... `rename()` returns zero if the file was renamed successfully. Otherwise, a nonzero value is returned.

ResetChannel()

Usage: ....
```
#include <system.h>
int ResetChannel(ptr);
channel *ptr;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `ResetChannel()` resets the channel pointed to by `ptr` in order to make it available for communication instructions.

Return values:.... `ResetChannel()` returns the old contents of the channel word pointed to by `ptr`.

Notes and remarks: This routine should be used when the user wants to reset a channel. It is also possible to write `NotProcessP` into a channel by writing

```
        *( int *) &chan = NotProcessP;
```

where `chan` is a channel. The manipulation with pointer operators and typecasts is necessary to avoid the transmission of the value `NotProcessP` over the channel. This method is not recommended. It does not work on links, as the link hardware is not reset.

Resetting a channel which has a process waiting for it will effectively abort the waiting process, since its information is not stored anywhere in either the transputer's hardware nor in the system areas of the Par.C System. The contents of the channel can be used to reach the waiting process, but great care should be taken when working with this information. See the transputer reference manual for more details on the communication mechanism and on the de/scheduling mechanism of the transputer.

Example: ....
```
#include <transp.h>

...

struct Port    /* stucture containing a channel */
{
   channel Chan;    /* This channel is not automatically
                            initialized */
   int Status;
} *Out;

...
   /* dynamically create a channel */
   Out = ( struct Port *)malloc( sizeof( struct Port ) );
   ResetChannel( &Out->Chan ); /* initialize the channel */
   Out->Status = FREE;
                    /* initialize rest of the structure */

/* after this, channel is available for communication */
```

_ResetSystemTimers()

Usage: ....

```
#include <system.h>
void   _ResetSystemTimers(Start);
int Start;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `_ResetSystemTimers()` sets both low and high priority system timers to the value indicated by `start`.

Return values: .... none

Notes and remarks: It is not necessary to start the timers, because the Par.C system sets their values to `MostNeg` just before calling `main()`. Resetting the timer may influence other processes, and is therefore not recommended. The timer can be read with the function `clock()`.

See also: .... `clock()`

rewind()

Usage: ....

```
#include <stdio.h>
void rewind(stream);
FILE *stream;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `rewind()` sets the file position indicator for the stream pointed to by `stream` to the beginning of the file. The end-of-file and error indicators for the specified stream are cleared.

Return values: ... none

Notes and remarks: The effect of `rewind()` can also be obtained by a call to `fseek()` with the offset specified as zero bytes from the beginning of the file, and a subsequent call to `clearerr()` for this file.

See also: ....

`fseek()`
`clearerr()`

Run()

Usage: ....

```
#include <stdlib.h>
int     Run(Linkno, Progfilename, Arguments)
int     Linkno;
char    *Progfilename, *Arguments;
```

Availability: ...... Par.C System

Library: .... `run.lib`

Description: ..... `Run()` reads the file `Progfilename` (containing an executable Par.C program) from the host and sends it over link `Linkno` to a neighbouring transputer, where it is started. The string `Arguments` is passed to the started program.

Calling `Run()` on the first transputer is comparable with invoking the command run Progfilename Arguments on the host machine.

The program calling `Run()` should be linked with `run.lib` and `io.lib`. The program(s) to be booted into neighbouring transputers should be linked with `noio.lib`.

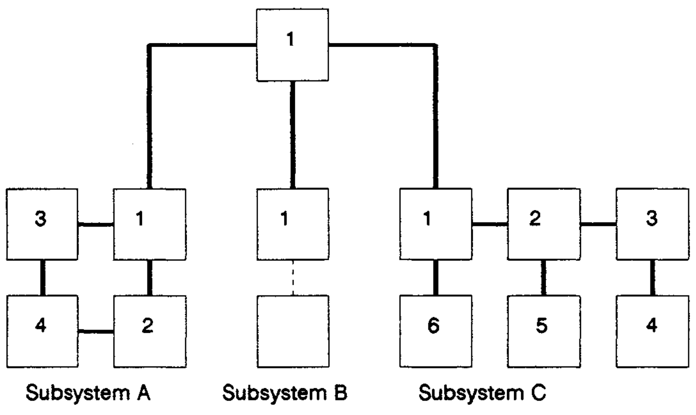
Detailed description

`Run()` can only be called on the first transputer in the network. It is used to boot a subsystem of transputers, connected to one of the links of this first transputer, with an executable Par.C program. Although that program is a complete Par.C program in itself, it will be referred to as a subprogram here.

The first transputer acts as a host to the subprogram. To avoid complications in the explanation, this transputer will be referred to as pseudo-host. The only difference between invoking RUN on the host machine and calling `Run()` on the first transputer is that a subprogram communicates with the pseudo-host through user-defined message passer routines (no file I/O system is available in the subprogram).

With `Run()`, a transputer system can be divided into a number of subsystems each executing a different subprogram. Since the transputer has four links and one of these is in use as interface to the host system, the maximum number of subsystems/subprograms is three (not counting the program running on the first transputer which initiates these three subprograms). An example is given in the following figure:

Example: .... In this example, three subsystems have been booted, each through one of the available links on transputer #1. Since each of the subprograms behaves exactly as if it was started directly from the host machine, the transputers in each subsystem are numbered again starting from 1.



Three independant subsystems

When the subprogram to be loaded onto a subsystem is linked with b.lib, a multi-transputer program will be produced, which is loaded onto all available transputers. This is exemplified in subsystems A and C above. When a subprogram is linked with a.lib only one transputer will be booted. This is shown in subsystem B, where one more transputer is available but not used by the subprogram.

Restrictions:

`Run()` resides in `run.lib` which replaces the ordinary single and multi transputer boot libraries `a.lib` and `b.lib`. Since `Run()` reads a file to load onto a subsystem, the program calling `Run()` must be linked with `io.lib`.

All subprograms must be linked with `noio.lib`, since no file server is installed on the pseudo-host. This also implies that no program calling `Run()` can be loaded onto a subsystem: the use of this function is restricted to the transputer immediately connected to the host.

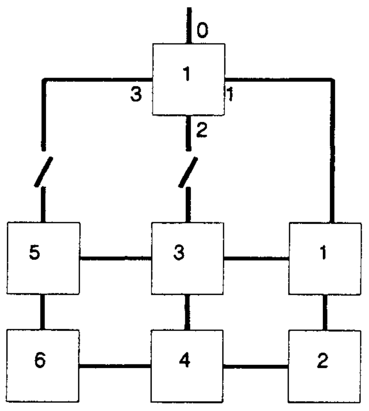
Return values: ... `Run()` returns zero if successful, otherwise one of the following error codes is returned:

`E_RSLFMT4`	Format of resident library for T4 corrupted.
`E_RSLFMT8`	Format of resident library for T8 corrupted.
`E_NOREALT4`	Unable to find/open resident library for T4.
`E_NOREALT8`	Unable to find/open resident library for T8.
`E_CANTALLOC`	Memory allocation failure on pseudo-host.

`E_NOSUFI`	Unable to find/open the indicated ProgFilename.
`E_TIMEOUT`	Timeout on communication to subsystem.
`E_LINKNO`	Attempt to access a non-existing link.
`E_CANTSEND`	Unable to send subprogram over the indicated link.
`E_CANTRECV`	Unable to receive from the indicated link.
`E_UNKREQ`	Unknown request code received from subsystem.

Also, a number of errors may occur in the file I/O between host and pseudo-host. See `errno.h` for a full list of error codes.

Notes and remarks: When booting a subsystem which is connected to a transputer already booted (including possibly the pseudo-host), care should be taken not to access the interconnecting link from the other side while `Run()` is executing. This is exemplified in the figure below:

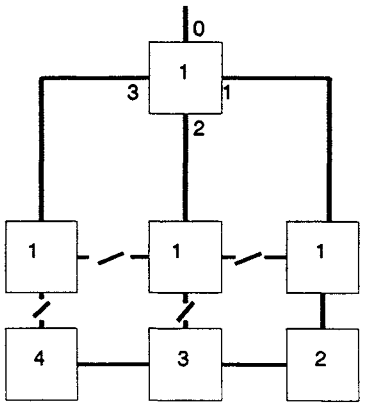


Multiple access to subsystem

In this example, the pseudo-host is connected to the subsystem via three links. When loading a multi-transputer subprogram through link 1, all transputers will be booted and the interconnections from transputers 3 and 5 back to the pseudo-host will also be probed by the network analyser. To prevent errors, links 2 and 3 of the pseudo-host should not be accessed in either direction during execution of `Run()`. This is indicated with the slash character '/'.

It is also possible to load different subprograms onto different parts of the same subsystem, but only to a limited extent. Using the same figure as in the previous note this can be explained as follows.

In this case, all transputers connected to the pseudo-host are again interconnected. Still, they can be divided into at most three subsystems by loading different subprograms through each of the three links going into the system from the pseudo-host. Since a multi-transputer program takes all available transputers, two of the subprograms in the above case must be single-transputer programs, and must be loaded before the multi-transputer program is loaded through link 1. Again, no access to the interconnecting links between different subprograms should be made during loadtime.



Different programs on the same subnetwork

`Run()` can be used in combination with the `_entry()` facility to boot a subsystem consisting of transputers with internal memory only. For an explanation of the use of `_entry()` see the chapter on Special Topics in this manual. The `\examples\pi` directory included in the Par.C System software contains source code (the `PI` programs) as examples of this technique.

See also: .... `errno.h`

Chapter 6

`_entry()`

RunProcess()

Usage: .... #include < system.h >

```
void RunProcess(Process, Priority, Npar, ...);
void (*Process)();
int Priority;
int Npar;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `RunProcess()` causes a process to be created, which calls the function pointed to by `Process` and passes the required parameters to it. This process will run concurrently with the caller of `RunProcess()`.

`Priority` can have the values `HIGH_PRIORITY` and `LOW_PRIORITY` as defined in `transp.h`, and indicates the priority at which the process will be started. After starting the concurrent process, `RunProcess()` returns immediately.

Any number of parameters can be passed to the function named `Process`. The optional fourth and following parameters of `RunProcess()` will be passed as parameters to the indicated function.

`Npar` must hold the size of the parameter list, which will mostly be the exact number of parameters, with the exception of longs, floats and doubles. Every long, float or double counts for 2 parameters.

The concurrent process will have its own workspace. When `Process()` returns the process is deactivated and its workspace deallocated.

Return values: .... none

Notes and remarks: `RunProcess()` offers an alternative to the **par** statement. These are the differences:

- `RunProcess()` can only start a piece of code that is organised as a function.
- `RunProcess()` can start processes at the high priority level.

- `RunProcess()` offers no automatic synchronisation mechanism: no parent process will wait for the started function to finish. It could even go on running after the main program has finished (which is not recommended).

See also: .... Chapter 4 (Parallel C)

Example: ....

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

#define PROCNR 20

void print_message(i,sync)
int i;
channel *sync;
{
   printf("This is process %d\n",i);
   *sync = 1;      /* signal that I am ready */
}

void main()
{
   int i, j;
   channel syncs[PROCNR];
   int ready[PROCNR];

   for (i=0; i<PROCNR; i++) { /* start PROCNR processes */
       RunProcess(print_message,LOW_PRIORITY,2,i,&syncs[i]);
       ready[i] = 0;
   }

   for (i=0; i<PROCNR; i++)
       /* wait PROCNR times for a ready signal */
       select {    /* look at running processes only */
           alt (j=0; j<PROCNR; j++) cond !ready[j]
               guard &syncs[j] :
                   ready[j] = syncs[j];
                   /* another process is ready */
       }
   /* at this point all processes which were started */
   /*  are ready */
}
```

scanf()

Usage: ....

```
#include <stdio.h>
int scanf(format, ...);
char *format;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `scanf()` reads input from the standard input stream, which is converted according to the format indicated in the format string pointed to by `format`, using the optional arguments following the format string to receive the input.

The format string may contain one or more whitespace characters, a number of ordinary characters (excluding the % character) and a number of conversion specification directives, each introduced with the % character.

A full description of the possible conversion specifications and flags is given in the description of the related `fscanf()`. The difference between `fscanf()` and `scanf()` is, that the latter reads from the standard input stream rather than from a stream indicated by the first argument.

Return values:.... `scanf()` returns the number of conversions or EOF if an error occurred before any conversion has been executed.

See also: .... `fscanf()`

SendLink()

Usage: ....

```
#include <system.h>
int SendLink(LinkNo, MessPtr, MessSize);
int LinkNo;
char *MessPtr;
int MessSize;
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: .... `SendLink()` will attempt to send a message of `Messsize` bytes to the output link indicated by `LinkNo`, reading the data from the array pointed to by `MessPtr`. Error codes are generated when the function is called with an illegal `LinkNo` or when the status of the link (as given in the `SYSTEM` structure) indicates I/O errors or the link not being connected.

Return values: ... `SendLink()` returns zero if the specified number of bytes were successfully sent. If the indicated link does not exist, the error code `E_LINKNO` is returned. If the linkstatus indicates a non-connected link or a link showing I/O errors, the error code `E_CANTSEND` is returned.

Notes and remarks: It is possible to perform link communication in other ways, e.g. by assigning link addresses to channel pointers and using the primitives for channel communication available in the Par.C System. Although system.h contains the macros `LINKOUT` and `LINKIN` for this purpose, this method of inter-processor communication is not recommended, because future expansions of the Par.C system might use the links for low-level message passing and internal system communications, or for multi-processor file I/O. The functions `RecvLink()`, `RecvLinkOrFail()`, `SendLink()` and `SendLinkOrFail()` are guaranteed to work, also in later versions of the Par.C System.

See also: ....

`RecvLinkOrFail()`
`RecvLink()`
`SendLinkOrFail()`

SendLinkOrFail()

Usage: ....
```
#include <system.h>
#include <time.h>
int SendLinkOrFail(LinkNo, MessPtr, MessSize, TimeOut);
int LinkNo;
char *MessPtr;
int MessSize;
clock_t TimeOut;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `SendLinkOrFail()` will attempt to send a message of `MessSize` bytes to the output link indexed by `LinkNo`, reading the data from the array pointed to by `MessPtr`. If this is not accomplished within `TimeOut` clockticks, the send is aborted and the function will return the non-zero value `E_TIMEOUT`. Error codes are also generated when the function is called with an illegal `LinkNo` or when the status of the link (as given in the `SYSTEM` structure) indicates I/O errors or the link not being connected.

Return values:.... `SendLinkOrFail()` returns zero if the specified number of bytes were successfully sent within the specified time. If the indexed link does not exist, the error code `E_LINKNO` is returned. If the linkstatus indicates a non-connected link or a link showing I/O errors, the error code `E_CANTSEND` is returned. If the send operation did not succeed within the specified time, the error code `E_TIMEOUT` is returned.

Notes and remarks: It is possible to perform link communication in other ways, e.g. by assigning link addresses to channel pointers and using the primitives for channel communication available in the Par.C System. Although `system.h` contains the macros `LINKOUT` and `LINKIN` for this purpose, this method of inter-processor communication is not recommended, because future expansions of the Par.C system might use the links for low-level message passing and internal system communications, or for multi-processor file I/O. The functions `RecvLink()`, `RecvLinkOrFail()`, `SendLink()` and `SendLinkOrFail()` are guaranteed to work, also in later versions of the Par.C System.

See also: ....

```
RecvLink()
RecvLinkOrFail()
SendLink()
```

Example: ....

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

int data[4] = { 10, 20, 30, 40 };
#define TIME_OUT   4000

void main()
{
   SYSTEM sysinfo;
   int myId, hostlink;
   int link, number;
   GetSysInfo(&sysinfo);
   myId = sysinfo.Tn;
   hostlink = sysinfo.HostLinkno;
   if(myId == 1){
       par(link = 0; link  4; link++){
           if(link != hostlink){
               SendLinkOrFail(link, &data[link],
                                sizeof(int), TIME_OUT);
               printf("send data %d down link %d\n",
                                data[link], link);
               RecvLinkOrFail(link, &data[link],
                                sizeof(int), TIME_OUT);
               printf("received result %d from link %d\n",
                                data[link], link);
           }
       }
       printf("finished test of Send-/Recv-LinkOrFail()\n");
   }else{
       RecvLinkOrFail(hostlink, &number, sizeof(int),
                                TIME_OUT);
       number = number + myId;
       SendLinkOrFail(hostlink, &number, sizeof(int),
                                TIME_OUT);
   }
}
```

setbuf()

Usage: . . . .

```
#include <stdio.h>
void setbuf(stream, buf);
FILE *stream;
char *buf;
```

Availability: . . . . . . Draft Proposed ANSI C

Library: . . . . `io.lib`

Description: . . . . . `setbuf()` causes the stream pointed to by the first argument to be buffered, using the buffer pointed to by `buf`. If `buf` is the `NULL` pointer, buffering is switched off.

Return values: . . . . `setbuf()` returns zero on success, and a nonzero value if the attempt to associate the specified stream with the buffer was unsuccessful.

See also: . . . . `setvbuf()`

_SetErr()

Usage: ....
```
#include <system.h>
int _SetErr();
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: .... `_SetErr()` sets the transputer's error flag.

Return values: .. `_SetErr()` returns a nonzero value.

Notes & remarks: . Note that the setting of the transputer's error flag will cause the processor to halt when the halt-on-error flag is also set.

See also: ....
```
_TestErr()
_TestClrErr()
_SetHaltErr()
_ClrHaltErr()
```

_SetFunStack()

Usage: ....
```
#include <system.h>
int _SetFunStack(size);
int size;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `_SetFunStack()` sets the basic size for function stacks to the value of `size`, and returns the previous value. It is used to determine the size of newly allocated stacks for function calls. This and related routines can be used to optimize program performance by tuning memory usage for stacks. Using too small stacks degrades function calling speed, using too large stacks wastes memory space. For an explanation on stack handling during runtime see the paragraph "Memory usage in Par.C programs" in chapter 6.

Return values:.... The previous basic function stacksize.

Notes and remarks: Controlling stackspace is global. During parallel processing some construct (e.g. using semaphores) should be employed when one wants to optimise for just a few function calls.

See also: ....
```
_GetFunStack()
_GetParStack()
_SetParStack()
```

_SetHaltErr()

Usage: ....
```
#include <system.h>
void _SetHaltErr();
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: .... `_SetHaltErr()` sets the transputer's halt-on-error flag. This will cause the transputer to halt when the error flag is set. The flag is off by default.

Return values: .. none

See also: ....
```
_SetErr()
_TestErr()
_TestClrErr()
_ClrHaltErr()
```

_SetParStack()

Usage: ....

```
#include <system.h>
int _SetParStack(size);
int size;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `_SetParStack()` sets the size for workspaces for new processes to the value of size. This only concerns the size of the initial stack allocated for each concurrent process.

Return value: .... The previous workspace size.

Notes and remarks: `size` is measured in words, not bytes.

Controlling stackspace is done on a global level. During parallel processing some construct (e.g. semaphores) should be employed when one wants to adjust the size for just a few par constructs.

Setting the value small causes newly created processes to allocate a new stack quite often when subroutine nesting gets deeper, while setting the value too large may waist memory. A small value is useful when many processes are started.

See also: ....

```
_GetFunStack()
_GetParStack()
_SetParStack()
```

SetPriority()

Usage: ....

```
#include <system.h>
int SetPriority(p);
int p;
```

Availability: ..... Par.C System

Library:......... `a.lib` and `b.lib`

Description: .... `SetPriority()` forces the current process to resume execution at the priority level indicated by `p`. The value of `p` may be set to zero for high priority and nonzero for low priority.

Return values: ... `SetPriority()` returns the priority level of the process before the function call was made.

Notes & remarks:. Processes running at the high priority level are discouraged in the Par.C System. It is our view and experience that processes running at high priority tend to slow down rather than speed up a program. However, timecritical and short-running tasks may be programmed to run on the high priority level to guarantee a short interrupt latency. In the Par.C System, whenever a high priority process uses resources like memory management or file I/O, it is forced to low priority until the call has been answered. The programmer should see to it, that tasks involving resources are handled outside of the timecritical regions of a process, or taken over by less timecritical processes (i.e. processes to buffer I/O). Of the routines contained in the Par.C System itself, only those involved in event-handling make use of the high priority level.

See also: .... `Priority()`

Example: ....

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

void calc()
{
   int i;
   double a = 3.0, b = 4.0;
   for (i=0; i<10000; i++) { a = a/b; b = b/a; }
}

void main()
{
   int i;
   par (i=0; i<10; i++)
   {{
       if (i%4 == 0) SetPriority(HIGH_PRIORITY);
       calc();
       printf("Ready : %d  priority = %d\n", i, Priority());
   }}
}
```

setvbuf()

Usage: ....

```
#include <stdio.h>
void setvbuf(stream, buf, mode, size);
FILE *stream;
char *buf;
int mode;
size_t size;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `setvbuf()` causes the stream pointed to by the first argument to be buffered, using the specifications given. The `mode` argument indicates the type of buffering and may have the value `_IONBF` (no buffering), `_IOLBF` (line buffering) or `_IOFBF` (full buffering). If `buf` is the `NULL` pointer and `mode` is not set to `_IONBF`, an automatically allocated buffer of size `size` is used. Otherwise, the character array `buf` points to is used. This array should have the size indicated by the last argument.

Return values: ... `setvbuf()` returns zero on success, and a nonzero value if the attempt to associate the specified stream and buffer was unsuccessful, or if invalid values are given for `mode` or `size`.

See also: .... `setbuf()`

signal()

Usage: ....
```
#include <signal.h>
void (*signal(sig, fptr))();
int sig;
void (*fptr)();
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `signal()` provides a primitive exception handling mechanism to be used to influence the normal flow of the C program. `signal()` is used to associate a specific signal value to a defined signal handling function, which will then be executed whenever the specified signal is raised. After a signal is raised, but before the corresponding signal handling function is called, the signal trap handler is reset to `SIG_DFL`. This implies that after every time a certain signal is raised, `signal()` must be called to again associate the signal to the same function, if multiple invocations are desired. Re-associating a signal handling function to a certain signal can be done inside the function itself. The `signal()` function expects a signal identifier of type integer and a pointer to a signal handling function which should be associated to the specified signal identifier. Instead of the signal handling function pointer, two other values may be used as second argument: `SIG_DFL` will cause the default signal handling to be executed. `SIG_IGN` will cause the signal to be ignored. Signals can be raised in two different ways: synchronously by a call to `raise()`, and asynchronously by the system traps.

`ABORT_SIG` is raised when the program is aborted from the keyboard and *BREAK* handling is supported, or when the routine `abort()` is called. `EVENT_SIG` is raised when the event-pin of the transputer is asserted. `ALARM_SIG` is raised when the time delay, set with `alarm()` expires. Signals with values 1 to 15 can be defined and used by the program. Other signals (including the value 0) are reserved for system use and future extensions.

Return values: .... `signal()` will return the value of `fptr` for the previous call to `signal()` with `sig` as first argument. If `signal()` fails to associate the specified signal value to the signal handling function pointed to, the value `SIG_ERR` will be returned.

Notes & remarks: . `signal(EVENT_SIG, EventHandler)` will cause the function EventHandler to be started as high-priority process, as soon as the event-pin on the transputer is asserted. If low-priority processing is desired, the call should be:

```
signal(EVENT_SIG | LOW_PRIORITY, EventHandler);
```

see also: ....

```
raise()
alarm()
sleep()
_sleep()
```

Example: ....

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

#define MYSIG 1

void ShowSignal(sig)
int sig;
{
   static int evtimes = 0;
   /* number of times the Event pin was asserted */

   switch(sig)
   {
       case MYSIG:
           printf("Somebody raised me\n");
           signal(MYSIG,ShowSignal); /* reinstall myself */
           break;
       case EVENT_SIG:
           printf("Event occured %d time(s)\n",++evtimes);
           signal(EVENT_SIG,ShowSignal);
               /* reinstall handler for Event signal */
           break;
   }
}
```

```
void main()
{
   signal(MYSIG,ShowSignal);
       /* install handler for my signal */
   signal(EVENT_SIG,ShowSignal);
       /* install handler for his signal */
   for(;;)           /* repeat indefinitely */
   {
       raise(MYSIG);
       sleep(4);
   }
}
```

sin()

Usage: ....

```
#include <math.h>
double sin(x);
double x;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `math.lib` and `t8math.lib`

Description: ..... `sin()` computes the sine of `x`. The argument `x` is expected to have the type double.

Return values: ... `sin()` returns the value of the sine of `x`.

sinh()

Usage: ....

```
#include <math.h>
double sinh(x);
double x;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `math.lib` and `t8math.lib`

Description: ..... `sinh()` computes the hyperbolic sine of `x`. The argument `x` is expected to have the type double.

Return values: .... `sinh()` returns the value of the hyperbolic sine of `x`.

sleep()

Usage: ....

```
#include <time.h>
int sleep(n);
int n;
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: ..... `sleep()` causes the process to be suspended for `n` seconds. After the indicated number of seconds have expired, the process is scheduled for execution again. If `ALARM_SIG` is raised before this number of seconds have passed, the process will be "woken", i.e. it will be put back in the active list.

Return values: ... `sleep()` returns zero if the indicated time has expired. If the process was resumed as an effect of `ALARM_SIG` being raised, the returned value is the number of unslept seconds.

Notes and remarks: To set a process waiting for a number of clockticks instead of seconds `_sleep()` can be used.

See also: ....

```
_sleep()
wait()
raise()
alarm()
```

_sleep()

Usage: ....
```
#include <time.h>
int _sleep(ticks);
time_t ticks;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `_sleep()` causes the process to be suspended for `ticks` clockticks, unless `ALARM_SIG` is raised before this number of clockticks have passed, in which case the process will be "woken up", i.e. it will be put back in the active list.

Return values:.... `_sleep()` returns zero if the indicated time has expired. If the process was resumed as an effect of `ALARM_SIG` being raised, the returned value is the number of unslept clockticks.

Notes and remarks: To have a process waiting for a number of clockticks without the possibility of waking up through `ALARM_SIG` being raised, `wait()` can be used.

See also: ....
```
sleep()
wait()
raise()
alarm()
```

smalloc()

Usage: ....

```
#include <stdlib.h>
char *smalloc(size);
size_t size;
```

Availability: ..... Par.C System

Library: .... `a.lib` and `b.lib`

Description: ..... `smalloc()` causes a memory block of `size` bytes (plus some words of overhead) to be allocated and a pointer to the allocated block to be returned. If no block of at least `size` bytes could be allocated, the `NULL` pointer is returned.

The difference with `malloc()` is that `smalloc()` allocates memory blocks at the low end of memory, which starts with the fast on-chip RAM of the transputer. If the on-chip RAM is not already occupied by program segments (See the description of the -b switch of the Par.C linker), the variables placed in the allocated block will have considerably lower access times.

Return values: ... `smalloc()` returns a pointer to the data area of an allocated block of at least `size` bytes, if allocation was successful. Otherwise, the `NULL` pointer is returned.

Notes and remarks: Using more than the allocated bytes will cause internal information of the memory manager to be overwritten, and will inevitably cause the program to crash.

The method of allocation used by `smalloc()` is optimized for size, which implies that only `size` bytes + some words overhead are allocated. All blocks are aligned to word boundaries and the value returned is a pointer to the data area of the allocated block. There is no guarantee that the return value of `smalloc()` will indeed point to a block in the fast on-chip memory of the transputer. If no on-chip RAM is available, the call `smalloc()` will continue the search into the external memory, since the entire available memory is seen by the memory manager as one continuous area with no distinction between internal and external memory. `smalloc()` allocates blocks of memory by a "first fit" method, and starts the search for available blocks from the low end of memory.

The corresponding function `malloc()` can be used to allocate blocks starting from the far end of memory.

See also: .... `malloc()`

Example: ....

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

void main()
{
   char *Low, *High;

   High = malloc(8);
   Low = smalloc(8);
   free( High );
   free( Low );
   printf("
       Largest free block starts at 0x%p, size = %u bytes\n",
       Low, High - Low + 8 );
}
```

sprintf()

Usage: ....

```
#include <stdio.h>
int sprintf(s, format, ...);
char *s, *format;
```

Availability: ..... Draft Proposed ANSI C

Library:......... `std.lib`

Description: ..... `sprintf()` writes characters to the string pointed to by `s`, in the format indicated in the format string pointed to by `format`, using the optional arguments following the format string. The format string may contain zero or more ordinary characters which are copied unchanged to the indicated string and zero or more conversion specifications, resulting in one or more subsequent arguments being converted to printing format and written to the string. Each conversion specification is preceded by the character `%`. Writing the character % to the string is achieved by putting `%%` in the format string.

A full description of the possible conversion specifications and flags is given in the description of the related function `fprintf()`. The difference between `fprintf()` and `sprintf()` is, that the latter writes to a string rather than to a stream indicated by the first argument. When the terminating null character is encountered, this is treated as an end-of-file marker.

Return values: ... `sprintf()` returns the number of characters written to the string `s` (not counting the terminating null character), or a negative value if an error occurred.

Notes and remarks: `sprintf()` writes a terminating null character to the string `s` after the format string has been processed entirely. This terminating null character is not included in the count which is returned after successful termination of the function.

See also: ....

```
fprintf()
printf()
vsprintf()
```

sqrt()

Usage: ....
```
#include <math.h>
double sqrt(x);
double x;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `math.lib` and `t8math.lib`

Description: ..... `sqrt()` computes the square root of the value of `x`. A domain error will occur when the argument has a negative sign.

Return values:.... `sqrt()` returns the square root of the value of `x`.

srand()

Usage: ....

```
#include <stdlib.h>
void srand(x);
unsigned int x;
```

Availability: ..... Draft proposed ANSI C

Library: .... `std.lib`

Description: .... `srand()` initializes the sequence generated by `rand()` to a known state. The same sequence will be generated after `srand()` is called with the same argument.

Return values: .. none

Notes and remarks: When `rand()` is called from several concurrent processes, the use of `srand()` can not guarantee that the same sequence will be generated.

See also: .... `rand()`

Example: ....

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

void main()
{
   int i;

   srand(1200);
   for (i=0; i<6; i++)
       printf("%12u", rand());
   printf("\n");

   srand(1200);
   for (i=0; i<6; i++)
       printf("%12u", rand());
   printf("\n");
}
```

sscanf()

Usage: ....
```
#include <stdio.h>
int sscanf(s, format, ...);
char *s, *format;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `sscanf()` reads input from the string pointed to by the first argument, which is converted according tot the format indicated in the format string pointed to by the second argument, using the optional arguments following the format string to receive the input. The format string may contain one or more whitespace characters, a number of ordinary characters (excluding the `%` character) and a number of conversion specification directives, each introduced with the `%` character.

A full description of the possible conversion specifications and flags is given in the description of the related `fscanf()`. The difference between `fscanf()` and `sscanf()` is, that the latter reads from a string rather than from a stream indicated by the first argument.

Return values: .... `sscanf()` returns the number of successful conversions, or `EOF` if an error occurred before any conversion has been executed.

Notes and remarks: When the trailing null character of the string `s` is encountered, this is treated as an end-of-file marker.

See also: .... `fscanf()`

Example: ....

```
#include <stdio.h>

char input[] = "1 + 2 = three\n";

void main()
{
   int i;
   long l;
   char c1, c2, s[8];

   sscanf(input, "%d %c %ld %c %s", &i, &c1, &l, &c2, s);
   printf("int %d\nchar %c\nlong %ld\nchar %c\nstring %s\n",
           i, c1, l, c2, s);
}
```

_StopProcess()

Usage: ....
```
#include <transp.h>
void _StopProcess();
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `_StopProcess()` stops the current process. It can be used to terminate a process which was started by `RunProcess()`. Note however, that the workspace of this process will not be freed. If the current process was started by means of a par, this par will never end.

Return values:.... `_StopProcess()` returns no value.

strcat()

Usage: ....

```
#include <string.h>
char *strcat(s1, s2);
char *s1, *s2;
```

Availability: ..... Draft Proposed ANSI C

Library:......... `std.lib`

Description: ..... `strcat()` concatenates the strings `s1` and `s2` to form one new string `s1`. The null character terminating `s1` before `strcat()` is called is overwritten by the first character of `s2`.

Return values: .. `strcat()` returns a pointer to the resulting string `s1`.

See also: .... `strncat()`

strchr()

Usage: . . . .

```
#include <string.h>
char *strchr(s, c);
char *s, c;
```

Availability: . . . . . . Draft Proposed ANSI C

Library: . . . . `std.lib`

Description: . . . . . `strchr()` searches for the first occurence of the character `c` in a string `s`. The string must be terminated by a null character. If the character `c` is found, a pointer to the first occurence of `c` is returned. Otherwise a `NULL` pointer is returned. Searching for a null character in string `s` will cause a pointer to the terminating null character of `s` to be returned.

Return values: . . . . `strchr()` returns a pointer to the first occurence of `c` in `s`, or a `NULL` pointer when `c` is not found.

See also: . . . . `strrchr()`

strcmp()

Usage: ....

```
#include <string.h>
int strcmp(s1, s2);
char *s1, *s2;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `strcmp()` takes the contents of strings `s1` and `s2` and compares them lexicographically. If the strings are equal, the value returned is zero. Otherwise, a non-zero value is returned, according to the lexicographical values of `s1` and `s2`.

Return values: .. `strcmp()` returns zero if strings `s1` and `s2` are identical. A non-zero value is returned if the strings are not identical: negative when the lexicographical value of `s1` is smaller, positive if the value of `s1` is larger than the value of `s2`.

See also: .... `strncmp()`
`memcmp()`

strcpy()

Usage: ....
```
#include <string.h>
char *strcpy(s1, s2);
char *s1, *s2;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `strcpy()` copies the contents of string `s2` to string `s1`, including the terminating null character, thereby overwriting the old contents of `s1`. The entire contents of `s2` is copied, even if `s1` is smaller than `s2`.

Return values: ... `strcpy()` returns a pointer to the first character of `s1`.

Notes and remarks Note that it is possible to corrupt a memory control block, and thus crashing the system, when the size of s2 exceeds the size of s1.

See also: .... `strncpy()`
`memcpy()`

strcspn()

Usage: ....

```
#include <string.h>
int *strcspn(s, set);
char *s, *set;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `strcspn()` searches the string `s` for the first occurence of a character that is contained in the string `set`, which is regarded as a set of characters rather than as a string. However, this second argument should also be terminated by a null character. The order of characters in this set, and whether or not there are duplicate characters in the set, is insignificant.

`strcspn()` returns a count of the first n characters in the string `s` which are not in the set of characters specified in the second argument. So, the first occurence of a character which is in the set will cause `strspn()` to return the number of skipped characters until then. If none of the characters of the string `s` are in the set of characters specified, the length of the string is returned, not counting the terminating null character. If `s` is the empty string, no characters can be found, and the return value will therefore be zero.

Return values: ... `strcspn()` returns the length of the longest initial segment of string `s` which contains no characters from the characters specified in `set`.

See also: ....

`strspn()`
`strpbrk()`
`strrpbrk()`

strerror()

Usage: ....
```
#include <string.h>
char *strerror(errnum);
int errnum;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `strerror()` maps `errnum` to an error message as found in `errno.h`, the error message is internally stored and can be accessed by means of the return value.

Return values:.... `strerror()` returns a pointer to an internal string containing the error message.

Notes and remarks: In case `errnum` cannot be mapped, a pointer to the string "error not found" is returned.

For parallel programming it is advisable to use `strerror_r()` which is a re-entrant version of `strerror()`.

See also: .... `errno.h`

`strerror_r()`

`perror()`

Example: ....
```
#include <string.h>
#include <stdio.h>
#include <errno.h>

void main()
{
   puts(strerror(E_UNKDEV));
}
```

strerror_r()

Usage: ....
```
#include <string.h>
char *strerror_r(message,errnum);
int errnum;
char *message;
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: ..... `strerror_r()` maps `errnum` to an error message as found in `errno.h`, the error message is stored in a string pointed to by `message`. The user should have declared at least `MAX_ERRMSG_LENGTH` characters string space.

Return values: ... `strerror_r()` returns `message`.

Notes and remarks: In case `errnum` cannot be mapped, the error message "error not found" is put in the user supplied string space.

For parallel programming this function is preferred to `strerror()` because it is re-entrant.

See also: .... `errno.h`

`perror()`

`strerr()`

strlen()

Usage: ....
```
#include <string.h>
size_t *strlen(s);
char *s;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `strlen()` returns the number of characters contained in the string `s`, not counting the terminating null character. The length of the empty string is zero.

Return values: .... `strlen()` returns the length of the string `s`.

strncat()

Usage: ....

```
#include <string.h>
char *strncat(s1, s2, n);
char *s1, *s2;
int n;
```

Availability: ..... Draft Proposed ANSI C

Library:......... `std.lib`

Description: ..... `strncat()` appends the first `n` characters in string `s2` to string `s1`, unless a null character is encountered in string `s2` before reaching the `n`-th character to be appended to string `s1`. The value of the resulting string `s1` is returned. The null character terminating `s1` before `strcat()` is called is overwritten by the first character of `s2`. The new string is terminated by a null character. If `n` is zero or has a negative value, no characters of string `s2` are appended to string `s1`.

Return values: ... `strncat()` returns a pointer to the resulting string `s1`.

See also: .... `strcat()`

strncmp()

Usage: ....

```
#include <string.h>
int strncmp(s1, s2, n);
char *s1, *s2;
int n;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `strncmp()` takes the first `n` characters of strings `s1` and `s2` and compares them lexicographically. If these character sequences are equal in value, zero is returned. Otherwise, a non-zero value is returned, according to the lexicographical values of the compared characters of `s1` and `s2`. If `n` has a zero or negative value, both strings are treated as empty strings and therefore are considered to be identical.

Return values: .... `strncmp()` returns zero if the first `n` characters of strings `s1` and `s2` are identical. A non-zero value is returned if these character sequences are not identical: negative when the lexicographical value of the first `n` characters of `s1` is smaller, positive if it is larger than the value of the first `n` characters of `s2`.

See also: .... `strcmp()`

`memcmp()`

strncpy()

Usage: ....

```
#include <string.h>
char *strncpy(s1, s2, n);
char *s1, *s2;
int n;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `strncpy()` copies the first `n` characters of string `s2` to string `s1`. If a null character is encountered before `n` characters of `s2` have been copied, null characters are written into `s1` until `n` characters have been written. If the length of string `s2` (excluding the terminating null character) exceeds the value of n, then `s1` will contain a truncated copy of string `s2`, and no terminating null character will be appended to `s1`.

Return values: ... `strncpy()` returns a pointer to the first character of `s1`.

See also: .... `strcpy()`

`memcpy()`

strpbrk()

Usage: . . . .

```
#include <string.h>
char *strpbrk(s, set);
char *s, *set;
```

Availability: . . . . . . Draft Proposed ANSI C

Library: . . . . `std.lib`

Description: . . . . . `strpbrk()` searches the string `s` for the first occurence of a character that is contained in the string `set`, which is regarded as a set of characters rather than as a string. However, this second argument should also be terminated by a null character. The order of characters in this set, and whether or not there are duplicate characters in the set, is insignificant.

`strpbrk()` returns a pointer to the first character in the string `s` which is in the set of characters specified in `set`. If none of the characters of the string are in the set of characters specified, the `NULL` pointer is returned. If `s` is the empty string, no characters can be found, and the return value will therefore be the `NULL` pointer.

Return values: . . . . `strpbrk()` returns a pointer to the first character in string `s` contained in the set of characters specified by `set`. A `NULL` pointer is returned if none of the characters in string `s` are contained in the `set` specified.

See also: . . . . `strcspn()`

`strrpbrk()`

Example: ....

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

char str[] = "The Par.C System by Parsec Developments\n";
char vowel[] = "aeiouy";

void main()
{
   char *result, *test = string;

   printf(str);
   while( ( result = strpbrk(test, vowel) ) != NULL )
   {
       int i = result - test;

       test = result + 1;
       while(i--) putchar(' ');
       putchar('^');
   }
   putchar('\n');
}
```

strrchr()

Usage: ....
```
#include <string.h>
char *strrchr(s, c);
char *s, c;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `strrchr()` searches for the last occurence of the character `c` in a string `s`. The string must be terminated by a null character. If the character `c` is found in string `s`, a pointer to the last occurence of `c` is returned. Otherwise the `NULL` pointer is returned. Searching for a null character in string `s` will cause a pointer to the terminating null character of the string to be returned.

Return values: .... `strrchr()` returns a pointer to the last occurence of `c` in string `s`, or the `NULL` pointer when `c` is not found.

See also: .... `strchr()`

strrpbrk()

Usage: ....

```
#include <string.h>
char *strrpbrk(s, set);
char *s, *set;
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: ..... `strrpbrk()` searches the string `s` for occurences of characters that are contained in the string `set`, which is regarded as a set of characters rather than as a string. However, this second argument should also be terminated by a null character. The order of characters in this set, and whether or not there are duplicate characters in the set, is insignificant.

`strrpbrk()` returns a pointer to the last character in the string `s` which is in the set of characters specified in the second argument. If none of the characters of the string `s` are in the set of characters specified, the `NULL` pointer is returned. If `s` is an empty string, no characters can be found, and the return value will therefore be the `NULL` pointer.

Return values: ... `strrpbrk()` returns a pointer to the last character in string `s` contained in the set of characters specified by `set`. The `NULL` pointer is returned if none of the characters in string `s` are contained in `set`.

See also: .... `strpbrk()`

`strcspn()`

strrpos()

Usage: . . . .

```
#include <string.h>
int strrpos(s, c)
char *s, c;
```

Availability: . . . . . . Par.C System

Library: . . . . `std.lib`

Description: . . . . . The function `strrpos()` searches the string s for the last occurence of the character `c`. If the character `c` is found in the string, the position of the last occurence is returned. If the character is not found, the value -1 is returned. Searching for a null character returns the position of the terminating null character, that is the length of the string, not the value -1.

Return values: . . . . The position of the last occurence of the character `c` in string `s`, or the value -1 if the character is not in the string.

strspn()

Usage: ....

```
#include <string.h>
int *strspn(s, set);
char *s, *set;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `strspn()` searches the string `s` for occurences of characters contained in the string `set`, which is regarded as a set of characters rather than as a string. However, this second argument should also be terminated by a null character. The order of characters in this set, and whether or not there are duplicate characters in the set, is insignificant.

`strspn()` returns a count of the first `n` characters in the string `s` which are all in the set of characters pointed to by `set`. The first occurence of a character which is not in the set will cause `strspn()` to return the number of skipped characters untill then. If all characters of the string `s` are in the set of characters specified, the length of `s` is returned, not counting the terminating null character. If `s` is the empty string, no matching characters can be found, and the return value will therefore be zero.

Return values: ... `strspn()` returns the length of the longest initial segment of string `s` which contains only characters which are contained in `set`.

See also: ....

`strcspn()`

`strpbrk()`

`strrpbrk()`

Example: ....

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

char str[] = "The Par.C System by Parsec Developments\n";
char vowel[] = "aeiouy";

void main()
{
   int skip, mark, count = 0;

   printf(string);

   while(count < strlen(string)){
       skip = strcspn(&string[count], vowel);
       count += skip;
       while(skip--) putchar(' ');

       mark = strspn(&string[count], vowel);
       count += mark;
       while(mark--) putchar('^');
   }
   putchar('\n');
}
```

strstr()

Usage: ....

```
#include <string.h>
char *strstr(src, sub);
char *src, *sub;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `strstr()` searches for the first occurence of the substring pointed to by `sub` in the string pointed to by `src`, and returns a pointer to this substring when found.

Return values: ... `strstr()` returns a pointer to the first occurence of the string `sub` in string `src`. The `NULL` pointer is returned if the substring has not been found in `src`.

See also: ....

`strtok()`
`strtok_r()`

strtod()

Usage: ....

```
#include <stdlib.h>
double strtod(s1, s2);
char *s1, **s2;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `strtod()` converts the string pointed to by `s1` to a floating-point number of type double. First the input string is decomposed into an initial sequence of zero or more whitespace characters (as specified by the `isspace()` facility), a subject sequence resembling a floating-point number representation, and a final string of one or more unrecognized characters including the trailing null character of the original string. The subject sequence is then converted to a floating-point number which is passed as return value.

The subject sequence is expected to contain an optional plus or minus sign, a sequence of digits optionally containing one decimal-point character and an optional exponent part (which in turn consists of the character 'e' or 'E' followed by an optional sign and two or more digits). If the first non-whitespace character in the original string is not a sign, a digit or a decimal-point character, the subject sequence remains empty and the returned double value is zero.

After succesful conversion, a pointer to the final subject sequence is stored in the object pointed to by `s2`, provided `s2` is not the NULL pointer.

Return values: .... `strtod()` returns the converted value of the floating-point number represented in `s1` and stores a pointer to the substring which could be identified as representing this floating-point number in `s2`. If conversion failed the returned value is zero.

See also: .... `atod()`

strtok()

Usage: ....

```
#include <string.h>
char *strtok(s, set);
char *s, *set;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `strtok()` is used to separate the string `s` into tokens separated by characters in the string `set`, which is used as a collection of characters rather than as a string. However, `set` should be terminated by a null character. `strtok()` must be called again for each new token in the string to be tokenised, passing a NULL pointer as first argument to obtain the next occurence of a character from the set of characters specified. This set of characters may be changed from one call of `strtok()` to another.

`strtok()` makes use of an internal state pointer, which points to the last token found in string `s`. With each call to `strtok()` which contains the NULL pointer as first argument, the string `s` is searched for characters in the set of characters specified, starting at the position in `s` pointed to by the internal state pointer. The first occurring character from `set` is replaced by a null character and the internal state pointer is returned. After this, the internal state pointer is adjusted to point to the first character in string `s` following the found character from `set`, which is replaced by a null character.

In this way, a string containing a number of tokens with specified token delimiters, is subdivided into a number of strings, each with a terminating null character, which are obtained one-by-one in successive calls to `strtok()`. Information on the progress of the tokenising process in string `s` is held in the internal state pointer, which is a variable local to `strtok()`.

Return values: ... `strtok()` returns a pointer to the first character of the substring, which is delimited by a character from the set of characters specified, or by the null character terminating string `s`. Each successive call to `strtok()` with the NULL pointer as first argument will cause a pointer to the next token in string `s` to be returned.

Notes and remarks: It follows from the explanation given above, that `strtok()` is non re-entrant by definition. This may cause serious problems when using `strtok()` in concurrent processes on the same transputer: when process 1 is descheduled before the tokenising process of string `s1` has finished, another process may call the same routine to tokenise another string, which causes the information on string `s1` to be erased.

To cope with this problem a special re-entrant version of the same function is added to the Par.C System, which copies the otherwise internal information on the tokenizing process to a variable local to the caller. See the description of `strtok_r()` for more details.

See also: .... `strtok_r()`

`strstr()`

strtok_r()

Usage: ....

```
#include <string.h>
char *strtok_r(s, state, set);
char *s, **state, *set;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `strtok_r()` is used to separate the string `s` into tokens separated by characters in the string `set`, which is used as a collection of characters rather than as a string. However, `set` should be terminated by a null character. `strtok_r()` must be called again for each new token in the string to be tokenised, passing a NULL pointer as first argument to obtain the next occurence of a character from the set of characters specified. This set of characters may be changed from one call of `strtok_r()` to another.

`strtok_r()` makes use of the state pointer pointed to by `state`, which points to the last token found in string `s`. With each succesive call to `strtok_r()` which contains the NULL pointer as first argument, the string `s` is searched for characters in the set of characters specified, starting at the position in `s` pointed to by `state`. The first occurring character from `set` is replaced by a null character and the state pointer is returned. After this, the state pointer is adjusted to point to the first character in string `s` following the found character from set, which is replaced by a null character.

In this way, a string containing a number of tokens with specified token delimiters, is subdivided into a number of strings, each with a terminating null character, which are obtained one-by-one in successive calls to `strtok_r()`.

Return values: ... `strtok_r()` returns a pointer to the first character of the substring, which is delimited by a character from the set of characters specified, or by the null character terminating string `s`. Each successive call to `strtok_r()` with the NULL pointer as first argument will cause a pointer to the next token in string `s` to be returned.

Notes and remarks: `strtok_r()` is the re-entrant replacement for the `strtok()` function as defined in the Draft Proposed ANSI C standard. The difference with `strtok()` is, that the internal state pointer used by the latter is replaced with a state pointer which is declared local to the routine calling `strtok_r()`. In this way, concurrent processes can call `strtok_r()` for different strings without interfering with each other's progress in tokenizing their respective strings. The state pointer should be declared as shown in the following example and its value should not be altered.

See also: .... `strtok()`

`strstr()`

Example: .... The following example shows the actual implementation of `strtok()` in the Par.C System.

```
#include <string.h>
#include <stddef.h>

char *strtok(str, delim)    /* non-reentrant function */
char *str;
char *delim;                /* the set of delimiters */
{
   static char *intern = NULL; /* internal state pointer */
   return strtok_r(str, &intern, delim);
}
```

The reentrant function can be used as outlined below.

```
#include <string.h>

void TokeniseString(str, delim);
char *str, *delim;
{
   char *nexttoken, *statepointer;

   ...
   while (EndOfStringReached == FALSE)
   {
       nexttoken = strtok_r(str, &statepointer, delim);
       ...
   }
}
```

strtol()

Usage: ....

```
#include <stdlib.h>
long int strtol(s1, s2, base);
char *s1, **s2;
int base;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `strtol()` converts the string pointed to by `s1` to a decimal number of type long integer. First the input string is decomposed into an initial sequence of zero or more whitespace characters (as specified by the `isspace()` facility), a subject sequence resembling an integer in some radix representation as determined by the value of `base`, and a final string of one or more unrecognized characters including the terminating null character of the original string. The subject sequence is then converted to a decimal number of type long integer which is passed as return value.

If the value of `base` is zero, the format of the subject sequence is expected to represent the octal notation of integers. If the value of `base` is between 2 and 36 the expected format of the subject sequence is an optional plus or minus sign, followed by a sequence of digits and letters. The letters a (or A) through z (or Z) are ascribed the values 10 through 35 and only letters whose ascribed value is less than that of `base` are recognized to be in the subject sequence. If the value of `base` is 16, the character sequence '0x' or '0X' is allowed to precede the sequence of digits and letters.

After successful conversion, a pointer to the unparsed part of `s1` is stored in the object pointed to by `s2`, provided `s2` is not the `NULL` pointer.

Return values: ... `strtol()` returns the converted value of the integer represented in `s1` with radix `base` and stores a pointer to the substring which could be identified as representing this integer in `s2`. If conversion failed, the returned value is zero.

Notes and remarks: In the Par.C System `strol()` is defined as a macro expanding to a call to the generic ASCII to integer conversion routine `_ato()`, which is also described in this chapter. `strtol()` is defined in `stdlib.h` for compatibility reasons, and also in `stdcnv.h` for reasons of convenience and completeness.

See also: .... `_ato()`

`stdcnv.h`

strtoul()

Usage: ....

```
#include <stdlib.h>
long int strtoul(s1, s2, base);
char *s1, **s2;
int base;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `strtoul()` converts the string pointed to by `s1` to a decimal number of type unsigned long integer. For a complete description of the functioning of `strtoul()` see the description of `strtol()`, which is equivalent, except that latter returns a signed rather than an unsigned long integer.

Return values: ... `strtoul()` returns the converted value of the integer represented in `s1` with radix `base` and stores a pointer to the unparsed part of the string in `s2`. If conversion failed the returned value is zero.

Notes and remarks: In the Par.C System `stroul()` is defined as a macro expanding to a call to the generic ASCII to integer conversion routine `_ato()`, which is also described in this chapter. `strtoul()` is defined in `stdlib.h` for compatibility reasons, and also in `stdcnv.h` for reasons of convenience and completeness.

See also: .... `_ato()`

`stdcnv.h`

system()

Usage: . . . .
```
#include <stdlib.h>
int system(cmdstring);
const char *cmdstring;
```

Availability: . . . . . . Draft Proposed ANSI C

Library: . . . . `io.lib`

Description: . . . . . `system()` passes the string pointed to by `cmdstring` to the host environment where the server executes a comparable function. A command processor (shell) is invoked at the host which interprets the command in the string. Which command interpreters are available, is host dependant. A `NULL` pointer may be used for `cmdstring` to inquire whether a command processor exists.

Return values: . . . . `system()` returns the exitcode of the invoked command processor and is host dependant. If the argument is a `NULL` pointer, `system()` returns non-zero when there is a command processor available.

Example: . . . .
```
#include <stdlib.h>

void main()
{
   if(system(NULL) != 0)
       system("dir");
}
```

tan()

Usage: ....

```
#include <math.h>
double tan(x);
double x;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `math.lib and t8math.lib`

Description: ..... `tan()` computes the tangent of `x`. The argument `x` is expected to have the type double.

Return values: ... `tan()` returns the value of the tangent of `x`.

tanh()

Usage: ....
```
#include <math.h>
double tanh(x);
double x;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `math.lib and t8math.lib`

Description: ..... `tanh()` computes the hyperbolic tangent of `x`. The argument `x` is expected to have the type double.

Return values:.... `tanh()` returns the value of the hyperbolic tangent of `x`.

_TestClrErr()

Usage: ....

```
#include <system.h>
int _TestClrErr()
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: ..... `_TestClrErr()` checks whether the transputer error flag is set, and clears it.

Return values: ... `_TestClrErr()` returns zero if the flag was not set. Otherwise, a non-zero value is returned.

Notes and remarks: This routine can be used to check on arithmetic overflow in certain sections of the program. The example listed below illustrates this.

See also: ....

`_SetErr()`

`_TestErr()`

`_SetHaltOnErr()`

`_ClrHaltOnErr()`

Example: ....

```
...
_TestClrErr();          /* clear flag before section     */
DoSomeCalculations();   /* execute critical section      */
Error1 = _TestClrErr(); /* check and clear error flag    */
DoSomethingElse();      /* execute another section       */
Error2 = _TestErr();    /* check error flag, don't clear */
...
```

_TestErr()

Usage: . . . .
```
#include <system.h>
int _TesttErr();
```

Availability: . . . . . . Par.C System

Library: . . . . `std.lib`

Description: . . . . . `_TestErr()` checks whether the transputer's error flag was set, and preserves the error flag while checking.

Return values: . . . . `_TestErr()` returns zero if the error flag was not set. Otherwise, a non-zero value is returned.

See also: . . . . `_SetErr()`

`_TestClrErr()`

`_SetHaltErr()`

`_ClrHaltErr()`

_TicksPerSecond()

Usage: ....
```
#include <stdlib.h>
int _TicksPerSecond();
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: ..... `_TicksPerSecond()` is used to make a correct implementation of the macro `CLOCKS_PER_SEC` which is defined in Draft Proposed ANSI C as giving the number of clockticks per second on each particular machine. Since the transputer currently uses timers with different frequencies (one for each of the priority levels), and since processes can set their own priority level during runtime, the value of `CLOCKS_PER_SEC` can only be determined through this function call.

In view of portability and future compatibility it is considered wise to use the `CLOCKS_PER_SEC` macro instead of calling `_TicksPersecond()`.

See also: .... `stdlib.h`

time()

Usage: ....
```
#include <time.h>
time_t time(ptr);
time_t *ptr;
```

Availability: ...... Draft Proposed ANSI C

Libraries: .... `io.lib` and `noio.lib`

Description: ..... `time()` returns an encoded value indicating the current calendar time, and stores the value in the string pointed to by `ptr`, unless this is the NULL pointer. In the current version of the Par.C System, the encoding is such that the return value gives the number of seconds which have passed since the first day of March 1972 at 00:00:00 Greenwich Mean Time.

The environment variable TIMEZONE should be used to set the difference with GMT in hours and minutes for the local timezone. Zones east of Greenwich are indicated by a positive value, zones west of Greenwich must use a negative value. The format allowed for the TIMEZONE variable is: [-]H[:M], where 0 = < H < 12 and 0 = < M < 60.

If the variable SUMMERTIME is defined in the environment, regardless of its value, local time is assumed to differ 1 hour more (positive) from GMT and the daylight savings time indicator will be set in the tm structure.

Return values:.... `time()` returns an encoded value indicating the current calendar time. If `ptr` is not the NULL pointer, the same value is stored in the string `ptr`.

Notes and remarks: Daylight savings time is not supported for all routines, because this would involve large tables for all zones and all times, which would not be worth the amount of memory needed. Most time related routines in the Par.C System allow manipulation of daylight savings time, but require the programmer to set this variable in the time structure.

See also: .... `gmtime()`

`mktime()`

`localtime()`

Example: ....

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

void main()

{
   time_t       Now_Encoded, RefTime_Encoded;
   struct tm    Now_Decoded, RefTime_Decoded;


/* get encoded time, decode and print it */
   Now_Encoded = time(NULL);
   Now_Decoded = *localtime (&Now_Encoded);
   printf ("Time: %2d:%02d:%02d\n", Now_Decoded.tm_hour,
               Now_Decoded.tm_min, Now_Decoded.tm_sec);


/* copy Now to RefTime to get the date etc right */
   RefTime_Decoded = Now_Decoded;


/* Set RefTime to 06:11:04 */
   RefTime_Decoded.tm_hour = 6;
   RefTime_Decoded.tm_min  = 11;
   RefTime_Decoded.tm_sec  = 4;


/* encode it */
   RefTime_Encoded = mktime (&RefTime_Decoded);

/* print the time elapsed between RefTime and Now */
   printf(
       "Elapsed time since 06:11:04 today: %5.0f seconds\n",
       difftime(Now_Encoded, RefTime_Encoded));
}
```

..toa()

Usage: ....

```
#include <stdlib.h>
int ..toa(number, string);
[unsigned] [long] int number;
char *string;
```

Availability: ...... Par.C System

Library: .... ---

Description: ..... `..toa()` indicates a collection of number-to-ASCII conversion facilities, which are all implemented as macros expanding to a call to the generic conversion routine _toa(). The macros `..toa()` convert `number` to ASCII and store the result in the string pointed to by `string`. The following gives a list of the available macros and their functions. The type of `number` should correspond to the specifications in this list.

`itoa`	: integer to decimal ASCII string
`utoa`	: unsigned int to decimal ASCII string
`xtoa`	: integer to hexadecimal ASCII string
`Xtoa`	: integer to capitalised hexadecimal ASCII string
`otoa`	: int to octal ASCII string
`btoa`	: int to binary ASCII string
`ltoa`	: long to decimal ASCII string
`ultoa`	: unsigned long to decimal ASCII string
`xltoa`	: long to hexadecimal ASCII string
`Xltoa`	: long to capitalised hexadecimal ASCII string
`oltoa`	: long to octal ASCII string
`bltoa`	: long to binary ASCII string

Return values: .... All `..toa()` functions return the length of the converted string.

See also: .... `_toa()`

`stdcnv.h`

_toa()

Usage: ....

```
#include <stdlib.h>
int _toa(string, control, input);
char *string;
int control;
[unsigned] [long] int input;
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: ..... `_toa()` is a generic number-to-ASCII conversion function. `string` points to a character array in which the result of the conversion will be stored, after which a null character will be appended. It is the programmer's responsibility to allocate enough space. The worst case is long binary conversion, which may consume up to 65 characters (including the terminating null cahracter).

`control` specifies the conversion type and consist of a bitwise or of the radix and additional bitfields. The radix may be in the range 2-36. A radix less then 2 results in no conversion taking place and `string` being empty. A radix greater than 36 results in the possibility of non-alphanumeric characters being inserted.

The bitmasks given in `stdcnv.h` (which is included in `stdlib.h`), further specify the conversion:

`_CV_LONG`	: input is to be considered 64 bits long.
`_CV_LOWER`	: use lowercase characters for digits greater than 9.
`_CV_SIGNED`	: perform signed conversion.

Return values: ... `_toa()` returns the number of characters inserted in the string pointed to by `string`, excluding the terminating null character.

See also: .... `..toa()`

toascii()

Usage: ....
```
#include <ctype.h>
int toascii(i);
int i;
```

Availability: ...... Draft Proposed ANSI C

Library: .... ---

Description: ..... `toascii()` converts `i` to a value in the range of ASCII coded characters, i.e. a value between 0 and 0x7F. All bits higher than the seven lower-order bits are set to 0. The ASCII coded character set can be found in the appendices.

Return values: .... `toascii()` returns an ASCII value for `i` by truncating the input value in order to let it fall in the range of the ASCII character set. If `i` was in this range already, the value is returned unchanged.

Notes and remarks: `toascii()` is implemented as a macro.

See also: .... `isascii()`

toint()

Usage: ....

```
#include <ctype.h>
int toint(c);
char c;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Return values: ... `toint()` returns the decimal value of the hexadecimal character `c`. If `c` is not a hexadecimal digit, a zero value is returned.

See also: .... `isxdigit()`

tolower()

Usage: ....
```
#include <ctype.h>
int tolower(c);
char c;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Return values: .... If `c` is an upper-case letter, `tolower()` returns the corresponding lower-case letter. Otherwise, the value of `c` is returned unchanged.

See also: .... `_tolower()`

`toupper()`

`islower()`

_tolower()

Usage: ....

```
#include <ctype.h>
int _tolower(c);
char c;
```

Availability: ..... Par.C System

Library: .... ---

Description: ..... This macro is a simple version of `tolower()` and will only work on letters. Other characters might produce unwanted results.

Return values: ... If `c` is an upper-case letter, `_tolower()` returns the corresponding lower-case letter. If `c` is lower-case, `c` is returned unchanged.

See also: .... `tolower()`

toupper()

Usage: ....
```
#include <ctype.h>
int toupper(c);
char c;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Return values:.... If `c` is a lower-case letter, `toupper()` returns the corresponding upper-case letter. Otherwise, the value of `c` is returned unchanged.

See also: .... `_toupper()`

`tolower()`

`isupper()`

_toupper()

Usage: ....
```
#include <ctype.h>
int _toupper(c);
char c;
```

Availability: ..... Par.C System

Library: .... ---

Description: ..... This macro is a simple version of `toupper()` and will only work on letters. Other characters might produce unwanted results.

Return values: ... If `c` is an lower-case letter, `_toupper()` returns the corresponding upper-case letter. If `c` is upper-case, `c` is returned unchanged.

See also: .... `tolower()`

ultoa()

Usage: . . . .

```
#include <stdcnv.h>
int ultoa(num, str)
long unsigned int num;
char *str;
```

Availability: . . . . . . Par.C System

Description: . . . . . The macro `ultoa()` converts the long unsigned integer `num` to its ASCII representation in decimal notation. The result is stored in the string pointed to by `str`. `str` should be sufficiently long to store the result of the conversion.

Return values: . . . . The length of the converted string.

See also: . . . . `_toa()`

utoa()

Usage: ....

```
#include <stdcnv.h>
int utoa(num, str)
unsigned int num;
char *str;
```

Availability: ..... Par.C System

Description: ..... The macro `utoa()` converts the unsigned integer `num` to its ASCII representation in decimal notation. The result is stored in the string pointed to by `str`. `str` should be sufficiently long to store the result of the conversion.

Return values: ... The length of the converted string.

See also: .... `_toa()`

ungetc()

Usage: ....
```
#include <stdio.h>
int ungetc(c, stream);
int c;
FILE *stream;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `io.lib`

Description: ..... `ungetc()` pushes the character specified by `c` and converted to an unsigned char back onto the input stream pointed to by `stream`. The effect is that reading the character from the stream is undone, and therefore the character is the first available character from the stream, provided that no functions will be called which change the file position pointer.

Return values: .... `ungetc()` returns an integer representing the character pushed back onto the input stream. If an error occurs `EOF` is returned.

See also: .... `fgetc()`

`getc()`

V()

Usage: ....
```
#include <stdlib.h>
void V(SemPtr);
semaphore *SemPtr;
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: ..... `V()` implements semaphores, together with the `P()` function and the `semaphore` type definition. `V()` operates on a user-defined semaphore and causes the next process waiting for the same semaphore to be scheduled for execution.

Semaphores can be used to have processes share resources or variables. In the Par.C System, semaphores are pointers and are used to implement a queue of processes waiting for the same semaphore. The queue is handled in the first-in-first-out way. Processes of different priorities are treated alike, i.e. processes running on high priority have no privilege in obtaining the semaphore.

Before using the `P()` and `V()` functions, a semaphore should be declared and initialised to contain the value `NULL`:

```
semaphore MySem = NULL;
P(&MySem);
...          /* critical section of this program */
V(&MySem);
```

When a process calls `P()` with MySem having the value `NULL`, the process is given access to its critical section and the semaphore is set to indicate the active process. The critical section is terminated when this active process calls `V()`. When another process is inside its critical section guarded by `MySem`, the process calling `P()` will be queued and temporarily stopped. The waiting process will resume execution after all preceding processes in the same queue have been scheduled, entered and exited their critical sections.

Notes and remarks: It is vital for correct functioning of semaphores that a critical section in a process is started with a `P()` operation and ended with a `V()` operation. Performing a `V()` operation without a corresponding previous `P()`

operation will void the protection of the resource or variable. This may cause loss of active processes.

Forgetting to call `v()` will cause all following code to be inside the critical section: other processes will never obtain access to the shared resources or variables and the program will probably deadlock as a result of this.

Processes running in a critical area are interruptable. The process is scheduled and de-scheduled as usual. However, the processes waiting for the semaphore will not interrupt this process before it releases it. Therefore, for the safe execution of a critical section every process that uses this section, should do so through `P()` and `v()`.

See also: .... `P()`

va_arg()

Usage: ....
```
#include <stdarg.h>
type va_arg(ap, type);
va_list ap;
```

Availability: ..... Draft Proposed ANSI C

Description: ..... The macro `va_arg()` provides, together with the type `va_list` and the macro's `va_start()` and `va_end()`, variable argument lists in function calls. `va_arg()` expands to the type and value of the next argument in the call. The state variable `ap` is advanced to indicate the next parameter in the list (if the end of the argument list has not been reached yet). `va_arg()` takes as arguments the current state variable and `type`, the type of the expected argument.

Return values: ... `va_arg()` returns the value of the next variable in the variable argument `ap` list with the type indicated in `type`.

Notes and remarks: The type `va_list` is defined in `stdarg.h`.

The implementation of the variable argument list facilities in the Par.C System makes use of a function `_va_arg()`, which is declared in `stdarg.h` but which is intern to the Par.C System and will not be described in this manual. `_va_arg()` resides in `std.lib`.

See also: .... `va_start()`

`va_end()`

Example: ....

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

void PrintVarArgs(format)
char *format;
{
   va_list ap;

   va_start(ap, format); /* initialise argument pointer */

   while(*format != '\0'){ /* continue up to end of format */
       switch(*format++){
           case 'I':               /* print an integer */
               printf("Integer %d\n", va_arg(ap, int) );
               break;
           case 'D':               /* print a double */
               printf("Double %e\n", va_arg(ap, double) );
               break;
       }
   }
   va_end(ap);          /* restore the argument pointer */
}

void main()
{
   PrintVarArgs("IDI", 12, 1.625, 15);
   PrintVarArgs("DI", 12345.6789e5, 123456789);
}
```

va_end()

Usage: ....
```
#include <stdarg.h>
void va_end(ap);
va_list ap;
```

Availability: ..... Draft Proposed ANSI C

Description: ..... The macro `va_end()` provides, together with the type `va_list` and the macro's `va_start()` and `va_arg()`, variable argument lists in function calls. `va_end()` should be called after the last variable in the argument list is read, and is intended to do cleaning-up of temporary variables etc. In the Par.C System, `va_end()` is defined with an empty body.

Return values: ... none

Notes and remarks: The type `va_list` is defined in `stdarg.h`.

The implementation of the variable argument list facilities in the Par.C System makes use of a function `_va_arg()`, which is declared in `stdarg.h` but which is intern to the Par.C System and will not be described in this manual. `_va_arg()` resides in `std.lib`.

See also: .... `va_start()`

`va_arg()`

va_start()

Usage:

```
#include <stdarg.h>
void va_start(ap, parm);
va_list ap;
identifier parm;
```

Availability: Draft Proposed ANSI C

Description: The macro `va_start()` provides, together with the type `va_list` and the macros `va_arg()` and `va_end()`, variable argument lists in function calls. `va_start()` should be called before the first variable in the variable argument list is read, to initialise the state variable `ap` to indicate the first argument in the variable argument list.

`va_start()` takes as its arguments the locally defined internal state pointer and the name of the last fixed parameter in the argument list.

Return values: none

Notes and remarks: The type `va_list` is defined in `stdarg.h`.

The implementation of the variable argument list facilities in the Par.C System makes use of a function `_va_arg()`, which is declared in `stdarg.h` but which is intern to the Par.C System and will not be described in this manual. `_va_arg()` resides in `std.lib`.

See also: `va_arg()`

`va_end()`

vfprintf()

Usage: ....

```
#include <stdio.h>
int vfprintf(stream, format, args);
FILE *stream;
char *format;
va_list args;
```

Availability: ..... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `vfprintf()` writes output to the output stream pointed to by the `stream`, in the format indicated in the format string pointed to by `format`, using the arguments given via the variable argument list indicated by `args`.

The format string may contain zero or more ordinary characters which are copied unchanged to the output stream and zero or more conversion specifications, resulting in one or more subsequent arguments being converted to printing format and written to the output stream. Each conversion specification is preceded by the character *%*. Writing the character *%* to the output stream is achieved by putting *%%* in the format string.

A full description of the conversion specifications and flags is given in the description of the related `fprintf()`. The difference between `fprintf()` and `vfprintf()` is that the latter takes its arguments from a variable argument list indicated by `args`, using `va_start()`, `va_arg()` and `va_end()` to get the arguments needed, rather than expecting them all to be named in the function call.

Return values: ... `vfprintf()` returns the number of characters written to the output stream, or a negative value if an error occurred.

See also: .... `fprintf()`

`stdarg.h`

vprintf()

Usage: ....
```
#include <stdio.h>
int vprintf(format, args);
char *format;
va_list args;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `vprintf()` writes output to the standard output stream in the format indicated in the format string pointed to by `format`, using the arguments given via the variable argument list indicated by `args`.

The format string may contain zero or more ordinary characters which are copied unchanged to the standard output stream and zero or more conversion specifications, resulting in one or more subsequent arguments being converted to printing format and written to the standard output stream. Each conversion specification is preceded by the character *%*. Writing the character *%* to the standard output stream is achieved by putting *%%* in the format string.

A full description of the conversion specifications and flags is given in the description of the related function `fprintf()`. The difference between `fprintf()` and `vprintf()` is that the latter writes to the standard output stream rather than to a stream indicated by an argument, and that the latter takes its arguments from a variable argument list indicated by `args`, using `va_start()`, `va_arg()` and `va_end()` to get the arguments needed, rather than expecting them all to be named in the function call.

Return values: .... The number of characters written to the standard output stream, or a negative value if an error occurred.

See also: .... `fprintf()`

`printf()`

`stdarg.h`

Example: ....
```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>

#define COUNT 25

int print_mess(format)
char *format;
{
   va_list ap;
   int result;

   va_start(ap, format);
   result = vprintf(format, ap);

   if(result < 0)
   {
       fprintf(stderr, "Unable to write to stdout!\n");
       exit(1);
   }

   return result;
}

void main()
{
   int i;

   printf("Concurrently printing %d messages\n\n", COUNT);
   par(i = 1; i <= COUNT; i++)
   {
       print_mess("Printing from process %d\n", i);
           /* one extra parameter */
   }
   print_mess("\nFinished with par\nBye now!\n");
           /* no  extra parameter */
}
```

vsprintf()

Usage: ....
```
#include <stdio.h>
int vsprintf(s, format, args);
char *s, *format;
va_list args;
```

Availability: ...... Draft Proposed ANSI C

Library: .... `std.lib`

Description: ..... `vsprintf()` writes characters to the string pointed to by `s`, in the format indicated in the format string pointed to by `format`, using the arguments given via the variable argument list indicated by `args`.

The format string may contain zero or more ordinary characters which are copied unchanged to the indicated string and zero or more conversion specifications, resulting in one or more subsequent arguments being converted to printing format and written to the string. Each conversion specification is preceded by the character *%*. Writing the character *%* to the string is achieved by putting *%%* in the format string.

A full description of the conversion specifications and flags is given in the description of the related function `fprintf()`. The differences between `fprintf()` and `vsprintf()` is that the latter writes to a string rather than to a stream indicated by an argument and that the latter takes its arguments from a variable argument list indicated by `args`, using `va_start()`, `va_arg()` and `va_end()` to get the arguments needed, rather than expecting them all to be named in the function call.

Return values: .... `vsprintf()` returns the number of characters written to the string `s` (not counting the terminating null character), or a negative value if an error occurred.

Notes and remarks: When the trailing null character of `s` is encountered, this is treated as an end-of-file marker.

vsprintf() writes a terminating null character to the string `s` after the format string has been processed entirely. This terminating null character is not included in the count which is returned after successful termination of the function.

See also: .... `fprintf()`

`sprintf()`

`stdarg.h`

wait()

Usage: ....
```
#include <time.h>
void wait(ticks);
clock_t ticks;
```

Availability: ...... Par.C System

Library: .... `std.lib`

Description: ..... `wait()` causes the process to be suspended for `ticks` clockticks. When the indicated number of clockticks have expired, the process is scheduled for execution again.

Return values:.... none

Notes and remarks: Calling `wait()` does not guarantee that the process will resume execution after exactly `ticks` clockticks, since other active processes may precede the process, once it has been put back in the active process list.

In contrast to `sleep()`, `wait()` is unconditional: processes using this function will not be resumed after `ALARM_SIG` has been raised.

The duration of one clocktick is dependant on the current priority. The macro `CLOCKS_PER_SEC`, defined in `time.h`, specifies the number of clockticks per second in a priority independant way.

See also: .... `sleep()`

`_sleep()`

`alarm()`

`signal()`

`raise()`

`time.h`

xltoa()

Usage: ....
```
#include <stdcnv.h>
int xltoa(num, str)
long int num;
char *str;
```

Availability: ..... Par.C System

Description: ..... The macro `xltoa()` converts the long integer `num` to its ASCII representation in hexadecimal notation. The digits 'a'-'f' are in lower case. The result is stored in the string pointed to by `str`. The string should be sufficiently long to store the result of the conversion.

Return values: ... The length of the converted string.

See also: .... `_toa()`

Xltoa()

Usage: . . . .
```
#include <stdcnv.h>
int Xltoa(num, str)
long int num;
char *str;
```

Availability: . . . . . . Par.C System

Description: . . . . . The macro `Xltoa()` converts the long integer `num` to its ASCII representation in hexadecimal notation. The digits 'A'-'F' are capitalized. The result is stored in the string pointed to by `str`. The string should be sufficiently long to store the result of the conversion.

Return values: . . . . The length of the converted string.

See also: . . . . `_toa()`

xtoa()

Usage: ....

```
#include <stdcnv.h>
int xtoa(num, str)
int num;
char *str;
```

Availability: ..... Par.C System

Description: ..... The macro `xtoa()` converts the integer `num` to its ASCII representation in hexadecimal notation. The digits 'a'-'f' are in lower case. The result is stored in the string pointed to by `str`. The string should be sufficiently long to store the result of the conversion.

Return values: ... The length of the converted string.

See also: .... `_toa()`

Xtoa()

Usage: ....
```
#include <stdcnv.h>
int Xtoa(num, str)
int num;
char *str;
```

Availability: ...... Par.C System

Description: ..... The macro `Xtoa()` converts the integer `num` to its ASCII representation in hexadecimal notation. The digits 'A'-'F' are capitalized. The result is stored in the string pointed to by `str`. The string should be sufficiently long to store the result of the conversion.

Return values:.... The length of the converted string.

See also: .... `_toa()`

yday()

Usage: ....

```
#include <time.h>
int yday(day, month, year);
int day, month, year;
```

Availability: ..... Par.C System

Library: .... `std.lib`

Description: ..... `yday()` checks the date built from `day`, `month` and `year` for validity and returns the day offset within the given year.

Return values: ... `yday()` returns the day number within the year, counted from the first of January. The first of January will return 1, while the 31th of December will return 365 (or 366 in a leap year). The routine will return 0, if the date is invalid or inconsistent.

See also: .... `DateToInt()`

`IntToDate()`

Examples: ...... Calls with inconsistent or invalid dates and so returning 0:

```
yday(1,12,1988)         - 0 <= month < 12
yday(1000,2,99)         - March has only 31 days
yday(31,3,1903)         - April only has 30 days
yday(29,1,1900)         - 1900 is not a leap year
```

Appendices

Differences with previous versions

Improvements and changes up to v1.2

Compiler

The compiler directive `#pragma fpu` caused incorrect results of logical operations on floating point numbers. This has been corrected.

Handling of floating point constants has been improved to result in a 1/2-bit accuracy (was 1 bit).

Conversion of negative integer constants to floating point values was done unsigned, this has been corrected.

The check on the remaining stack size at each function entry is generated in-line, which increases performance.

Linker

The functioning of the `-a` switch has been reversed: invoking the linker without `-a` causes word alignment of object modules. This has been done to avoid errors with floating point numbers.

Loader/Server

The `-i` option has been added, to initialise the memory of the transputer to zero, before loading a program.

The `-r` option has been added, to enable loading a program without first resetting the transputer system.

The `-d` option has been added, to have the server display messages tracing file I/O.

Runtime libraries

Memory management has been optimised considerably. Performance does not degrade anymore by allocating many blocks.

`fputc()` caused a buffer overflow on some occasions. This has been corrected.

System information is now only passed to the program on explicit request through a function call: `GetSysInfo()` and `GetNodeInfo()`.

`..toa()` and related functions have been redefined to conform to usage in standard C compiler systems. Although these functions are not part of the Draft Proposed ANSI C standard, the ordering of arguments for these macro's and the associated functions has been redefined to be closer to what one might expect.

The `atan()` function in `math.lib` returned incorrect results for an argument smaller than -1.0. This has been corrected.

`t8math.lib` has been added for enhanced performance on transputers of the T8XX series.

Error handling has been improved on a number of levels:

- All functions in the `t8math.lib` call `matherr()`, which has been improved to do more than just exit. See the description of `matherr()` for more information.
- `perror()`, `strerror()` and `strerror_r()` have been added to the libraries.
- Errors occurring during file I/O with the host system are now displayed as messages on the screen.
- Some fatal runtime errors (like OUT OF MEMORY) will cause a message to be displayed on the screen (only if the server is active when the error occurs).

These functions have been added to the libraries:

`_abort()`	`GetNodeInfo()`	`P()`
`_alarm()`	`_GetParStack()`	`PeekHost()`
`calloc()`	`GetSysInfo()`	`perror()`
`ceil()`	`InPort()`	`PokeHost()`
`Dir()`	`matherr()`	`realloc()`
`DirInit()`	`MemAvail()`	`Run()`
`(*ERROR)()`	`onexit()`	`_SetFunStack()`
`_GetFunStack()`	`OutPort()`	`_SetParStack()`

The following functions have been removed from the libraries:

- `leave()` has been removed because its functioning interfered with correct file handling on the host machine. See the description of `leave()` in this manual for more information on how to replace its functionality in various more secure ways.
- `curtime_r()` has been removed for lack of significance. See the description of `curtime_r()` for information on a replacing call to `localtime()`.

- `CRASH()` has been replaced by more sophisticated handling of fatal runtime errors. The routine was not meant to be user callable in the first place.

Improvements and changes since v1.2

Compiler

Code generation using `#pragma fpu` has been improved considerably. These improvements mainly apply to mixed type expressions.

Dividing and multiplying with the constants 2.0 or 0.5 rendered illegal instructions in the assembler output. This bug was only present in v1.2.

Function calls in complex expressions sometimes caused trouble: Some intermediate results would not be stored.

A call to a function of type double or float without parameters resulted in an Internal Error C8000.

Using arrays of enum resulted in an Internal Error in some instances. This has been fixed.

`float x=0.0;` Was compiled as : `float x=2.0;` This error was only present in v1.2 and has been removed.

Compound assignment to complex lvalues of type short resulted in an Internal Error. This has been fixed.

The code `<any_type> x[SIZE], *p; p = x + 2;` was compiled as if `x` were a character array, i.e. the constant was not multiplied with the size of `<any_type>`. This has been fixed.

Some inconveniences have been removed:

- The compiler did not recover properly from a missing closing parenthesis in a function header.
- An excess use of memory at compile time has been eliminated.

Runtime libraries

`qsort()` crashed in most cases. This has been fixed.

`scanf()`, `fscanf()` and `sscanf()` did not skip newline characters prior to a conversion and did not handle the `%hd` specifier for short conversion correctly either: if a pointer to an element of a short array was supplied, an adjacent element was overwritten. Both errors have been removed.

The function `pow()` from `t8math.lib` did not return correct values. This has been fixed.

`math.lib`, the math library which is suited for both T4XX and T8XX, has been speeded up considerably by using improved algorithms and writing large pieces of it in assembler. This has also brought down the size.

The handling of errors in `math.lib` and `t8math.lib` has been improved. The user can now install his own `matherr()` handler, which can access the arguments with which the math routine was called.

The function `modf()` has been changed to comply with the Draft Proposed ANSI standard. The second parameter was a pointer to integer in stead of a pointer to double.

Compiler / Runtime system

The conversion of floating point types to integral types was not consistent with the specification of the Draft Proposed ANSI standard, which states that the fraction of the floating point value should be dropped. Until now, the floating point value was rounded to the nearest integer. It has now been implemented according to the Draft Proposed ANSI standard. Note that this can alter the behaviour of existing programs.

Known bugs (and remedies)

Complex expressions

Using too complex expressions may cause the generation of erroneous code. This problem has been worked on, but it is not completely solved. Try to re-write the expression in a less complex form, using temporary variables when needed.

Note: .... The symptoms of this problem are the generation of the transputer instructions `stl`, `ldl` and `ldlp` with negative offsets and will in all cases be detected by the peephole optimizer, causing the compiler to exit with the following message:

```
C8000 Internal Error: Code Generator (EmitNoPHOpt)
```

This error is most likely to occur when using (single-length) floating-point variables in complex expressions.

Module alignment and FPU code

The use of `#pragma fpu` is liable to generate errors. When using floating point constants, the linker should be invoked without the `-a` option to align the object modules. The results when invoking the `-a` linker option are not guaranteed. This problem does not occur when linking a single object file, or when only the first object file contains these floating point constants.

Note: .... The functioning of the `-a` linker switch has been reversed, so invoking the linker with `-a` will not generate word-aligned object modules in the executable file.

Error and warning messages

Command line error messages

In the current version of the Par.C System, all components except the loader/server use the same routine to read and interpret the commandline options. This option routine may cause the following error messages to be displayed:

- `FATAL ERROR <message>`

  `Illegal commandline option`

- `FATAL ERROR <message>`

  `Illegal scriptfile option`

with < message > replaced by one of the following:

- `invalid script filename after @ option`
- `non-alphabetic option`
- `illegal option <character>`
- `not enough arguments or option as argument`
- `illegal sub-option <character>`
- `cannot open scriptfile <filename>`

Since these messages are obvious enough, no further explanation is given here. The error and warning messages generated by the Par.C compiler and linker are explained below.

Compiler error messages

The following error and warning messages may be generated by the compiler. The numbers indicate the class in which the errors and warnings fall.

Storage class

`C1101 Illegal storage class for global var. Extern assumed.`

`auto` or `register` was used as storage class for a global variable.

`C1102 Illegal storage class for parameter.`

A storage class other than `register` was used when declaring a parameter.

`C1103 Storage class specified twice.`

More than one storage class was specified in one declaration.

Array declaration

`C1201 Array size must be a positive integer.`

An array was declared with a negative or zero element count.

`C1202 Syntax error in declaration. ']' expected.`

The closing bracket in an array declaration was absent or misplaced.

`C1203 Syntax error in abstract declarator. ']' expected.`

A closing bracket in an abstract array declarator was absent or misplaced.

Initialisation

`C1301 Error in <name> initialiser. Constant expression expected.`

An attempt was made to initialise a global variable of type <name> with a non-constant expression.

`C1302 Cannot initialise global float/double.`

This message will only occur in versions of the Par.C System running on host systems which do not support IEEE floating point arithmetic. This makes it impossible to evaluate fp constants at compile time. In these cases, one should initialise the variable with an explicit assignment instead of using an initialiser.

`C1303 Illegal constant type for float/double initialiser.`

The constant supplied could not be converted to a `float/double`..

`C1304 Channels can not be initialised.`

The storage class associated with a `channel` is not used to hold a value, but an internal status. Therefore it is useless to initialise it to a user specified value. A pointer to a `channel` can be initialised to an address, for instance the address of a hardware link.

`C1305 Too many initialisers in array initialiser.`

The number of initialisers exceeded the specified element count in an array initialiser.

`C1306 Too many initialisers in structure initialiser.`

The number of initialisers exceeded the number of components of the designated structure in a structure initialiser.

`C1307 Array initialiser string too large.`

The number of characters in the string literal exceeded the specified element count.

`C1308 Error in array initialiser. Non-empty list needed.`

The list of initialisers in an array initialiser should contain at least one element.

`C1309 Only character arrays can be initialised with a string literal.`

An attempt was made to assign a quoted string to an array of a type differing from `char` or `unsigned char`.

`C1310 Function/void can not be initialised.`

An initialiser was used when declaring an object which can not be initialised. Initialisers can not be used with local arrays. Local arrays can only be initialised with explicit assignment statements.

`C1312 Parameters can not be initialised.`

A declaration of a parameter was followed by an initialiser.

`C1313 Initialiser not allowed in function declaration.`

An initialiser followed a function declaration.

`C1314 Variable size unknown. Complete type or initialiser needed.`

The compiler could not determine the size of a variable. This usually occurs when declaring types involving arrays when neither the array element count nor an initialiser are present.

`C1314 Syntax error in array initialiser. '{' expected.`
`C1315 Syntax error in structure initialiser. '{' expected.`

An array or structure initialiser should be enclosed in braces.

`C1316 Syntax error in array initialiser. Comma or '}' expected.`

`C1317 Syntax error in structure initialiser. Comma or '}' expected.`

After an array or structure initialiser element either a comma, indicating another initialiser is following, or a terminating bracket is required.

`C1318 Illegal kind of structure/union initialisation.`

Function declaration/definition

`C1401 Illegal function return type.`

A function was declared with an illegal type for the return value. Only simple types can be returned.

`C1402 Function definition not allowed here.`

Any function definition should be at top level, i.e. not local to another function. Possibly a comma or semicolon was omitted after a function declaration.

`C1403 Syntax error in parameter list. Comma or ')' expected.`

An identifier in a parameter list should be followed by either a (separating) comma or a (terminating) right parenthesis.

`C1404 Syntax error in parameter list. Identifier expected.`

The formal parameterlist in the function header should consist of identifiers, separated by commas and enclosed in parentheses.

`C1405 Syntax error in function definition. Function body or parameter declaration expected.`

The function header of a function definition should be followed by either the declaration of the formal parameters, or by the function body.

`C1406 Syntax error in function definition. Function body missing.`

A function definition should include a compound statement defining the function body. At least `{}` should be present.

`C1407 <name> is not a parameter.`

The identifier < name > was declared as formal parameter, but did not occur in the formal parameterlist.

`C1408 Parameter <name> declared more than once.`

A parameter was declared more than once in the formal parameter declarations.

`C1409 Parameter name <name> already used.`

The same identifier was used more than once in the formal parameterlist of a function definition.

`C1410 Illegal parameter type.`

A parameter in a function parameter list had an illegal type. Structures and unions can not be used as parameters.

General declaration

`C1501 Syntax error in declarator. ')' expected.`

An attempt to balance parenthesis in a declarator failed.

```
C1502 Syntax error in declarator. '*', '(' or identifier
expected.
```

Type specifiers and/or storages class specifier in declarations should be followed by one of these three tokens.

`C1503 Illegal parenthesized abstract declarator.`

The use of parentheses in an abstract declarator was not correct.

`C1504 Identifier not allowed in abstract declarator.`

In an abstract declarator (e.g. used in a cast or a `sizeof`) the use of an identifier is illegal. It should be omitted.

```
C1505 Illegal use of function type.
C1506 Illegal use of void type.
C1507 Illegal use of array type ('function returning
array').
C1508 Illegal use of function type ('function returning
function').
C1509 Illegal use of function type ('array of function').
C1510 Illegal use of undefined structure/union.
```

The type used is not a valid or meaningful type or is not valid in this context.

`C1511 Syntax error in declaration. Comma or semicolon expected.`

An identifier in a declaration was not followed by either a separating comma or a terminating semicolon.

Preprocessor

`C2101 Sizeof operator not allowed in preprocessor expression.`

The preprocessor cannot evaluate the `sizeof` operator in expressions.

`C2102 'defined' operator only allowed after #if or #elif.`

The `defined` operator can only be used in preprocessor directives `#if` and `#elif`

`C2103 Macro name expected.`

The `defined` operator should be followed by an identifier (a macro name).

`C2104 Too many formal macro arguments. Line ignored.`

The number of macro arguments exceeded the compiler limit in a macro definition.

`C2105 Illegal preprocessor command. Line ignored.`

The characters following the `#` were not recognized as a valid directive.

`C2106 Unbalanced #<name>. Ignored.`

The preprocessor encountered a 'closing' directive before encountering the corresponding 'opening' directive.

`C2107 Identifier expected as macro name. Line ignored.`

The directives `#define`, `#undef`, `#ifdef` and `#ifndef` should be followed by an identifier.

`C2108 Identifier expected as macro argument. Line ignored.`

Only identifiers are allowed as formal macro arguments.

`C2109 Formal macro argument used more than once. Line ignored.`

An identifier was used more than once in the formal parameter list of a macro definition.

`C2110 Filename missing. Line ignored.`
`C2111 Illegal filename delimiter. Line ignored.`

The directive `#include` should be followed by a filename, delimited by either angled brackets (`<...>`) or double quotes (`" ... "`).

`C2112 Constant expression expected. Line ignored.`

The directive `#if` should be followed by a constant expression.

`C2113 End of file reached expanding macro.`

An unexpected EOF was encountered during macro expansion. Check parentheses !

`C2114 Too many actual macro arguments.`
`C2115 Not enough actual macro arguments.`

The number of arguments in the use of a macro did not match the number of arguments in the macro definition.

`C2116 Enumeration constants not allowed in preprocessor expressions.`

An enumeration constant was used in a preprocessor expression.

`C2117 Illegal character: $XX`

An illegal character was found in the source file. The printed value is the hexadecimal representation of the character.

`C2118 Syntax error in macro definition. ',' or ')' expected.`

A formal macro argument should be followed by a comma or a right parenthesis.

Syntax errors in statements

`C3101 Case value already used.`

A constant value after a case statement was used more than once in the same switch statement.

`C3102 More than one default statement.`

More than one default statement was used in the same switch statement.

`C3103 Closing brace missing at end of file.`

A closing bracket '}' was omitted or misplaced. This is often caused by a preceding error and is due to the panic-mode error recovery.

```
C3104 Syntax error in <keyword> statement. '{' expected.
C3105 Syntax error in <keyword> statement. '}' expected.
C3106 Syntax error in <keyword> statement. '(' expected.
C3107 Syntax error in <keyword> statement. ')' expected.
C3108 Syntax error in <keyword> statement. ':' expected.
C3109 Syntax error in <keyword> statement. ';' expected.
```

The statement parser found that one of these delimiters was missing in the indicated statement.

```
C3110 'break' used outside while/do/for/switch/select.
```

`break` can only be used inside one of these statements.

```
C3111 'continue' used outside while/do/for.
```

`continue` can only be used inside one of these statements.

```
C3112 'return' in par statement not allowed.
```

A return from a subprocess would have disastrous results.

```
C3113 Syntax error in for-expressions. '(' expected.
```

`for` should be followed by a left parenthesis.

```
C3114 Syntax error in for-expressions. Semicolon expected.
```

The expressions in a `for` statement should be separated by a semicolon. The same holds for replicators in `alt` and `par`.

```
C3115 Syntax error in for-expressions. ')' expected.
```

The expressions in a `for` statement should be terminated by a right parenthesis. The same holds for replicators in `alt` and `par`.

```
C3116 'case' used outside switch statement.
```

`case` labels can only be used within a `switch` statement.

```
C3117 'default' used outside of switch statement.
```

`default` can only be used within a `switch` statement.

```
C3118 Syntax error in do-while statement. 'while' expected.
```

A `do` loop should be terminated with `while`.

`C3119 Syntax error in select statement. 'alt' expected.`

The first token following the first opening brace in a select statement should be `alt`

`C3120 Pointer to channel needed in guard specifier.`

`guard` should be followed by a pointer to a channel.

`C3121 Error in select statement. Illegal timeout specifier.`

A `timeout` alternative is only valid in a timed select statement, using `select within`.

`C3122 Error in select statement. 'timeout' specified more than once.`

`C3123 Error in select statement. 'timeout' in replicated alt.`

Only one `timeout` can be specified in a select statement.

`C3124 Error in alternative. No cond/guard/timeout specifier.`

`C3125 Syntax error in alternative. 'timeout', 'guard' or ':' expected.`

`C3126 Syntax error in alternative. ':' expected.`

Incorrect alternative. `alt` should be followed by `cond`, `guard` or `timeout`. `cond` should be followed by `timeout`, `guard` or a colon. The alternative should always be terminated by a colon.

`C3127 Illegal alternative.`

The alternative was not specified correctly.

`C3128 Syntax error in goto statement. Identifier expected as label.`

`goto` should be followed by an identifier.

`C3129 Unbalanced else.`

`else` was used without a corresponding `if`.

Errors in expressions

`C4101 Operand required.`

An identifier was used in an invalid context.

`C4102 Constant expression required.`

A non constant object was used in an expression which should evaluate to a constant value.

`C4103 Pointertype or arraytype required.`

A subscripting operator was used in combination with a variable of type different from pointer or array.

`C4104 Illegal type conversion.`

The types of the operands of an operator were incompatible with each other or with the specific operator.

`C4105 Subscripting of functions not allowed.`

A subscripting operation was attempted on a function.

`C4106 Function required for functioncall.`

A functioncall operator '`()`' was applied to an identifier which was declared, but not as a function.

`C4107 Structure/union type required.`

A direct selection operator '.' was applied to something other than a structure or a union.

`C4108 Not a component of this structure/union.`

The identifier following '.' or -> was not one of the components of the structure / union.

`C4109 Pointer to structure/union required.`

An indirect selection operator -> was applied to something other than a pointer to a structure/union.

`C4110 Lvalue required for <name> operator.`

The operators `++`, `--`, `&`, `=`, `+=`, `-=` and the other assignement operators should be applied to an lvalue, i.e. something evaluating to an address.

`C4111 Illegal use of structure/union type.`

An operator was illegally applied to a structure or union.

`C4112 Pointer required for indirection.`

The indirection operator `*` can only be applied to pointers.

`C4113 Illegal typecast.`

A cast operator was applied to an object of incompatible type.

`C4114 No list expression allowed in constant expression.`

`C4115 Assignment not allowed in constant expression.`

`C4116 Increment/decrement operator not allowed in constant expression.`

Constant expressions, like array element counts, can not contain a comma operator (','), an assignment ('=', '+=', '*=' ....) or a increment or decrement operator ('++', '--').

`C4117 : expected in conditional operator.`

A colon was missing in the conditional operator `?:`.

`C4118 ) expected.`

The expression parser expected a right parenthesis here.

`C4119 ] expected.`

The expression parser expected a right square bracket here.

`C4120 Cast to function type not allowed.`

A cast of any data type to function type is illegal.

`C4121 Component name expected.`

The operators '.' and -> should be followed by an identifier selecting the component of the structure / union.

`C4122 Cast to non-integers not allowed in constant expression.`

Only type casts to integer are allowed in constant expressions.

`C4123 Parameter missing.`

A comma in a function parameter list was not followed by a parameter.

`C4124 Operand expected.`

The expression parser expected an operand at this position.

Errors in identifiers

`C5101 Unresolved reference to structure/union <name>. Tag not defined.`

A structure / union tag was used in a structure / union definition, but could not be resolved.

`C5102 Unresolved reference to structure/union <name>. Not a structure cq union tag.`

The tag used in referencing a structure could not be identified as a structure / union tag.

`C5103 Conflict using symbol <name>.`

A symbol <name> was used in a way not compatible with previous references / definitions of the same symbol.

`C5104 Enumeration constant <name> already defined.`

The same identifier <name> was used more than once to define an enumeration constant.

`C5105 Tag <name> is not an enumeration tag.`
`C5106 Enumeration tag <name> not defined.`

The identifier <name> was used as a reference to an enumeration type which was not defined or not defined as an enumeration tag.

`C5107 Enumeration tag <name> already defined.`

A definition of an enumeration type used a tag of an earlier defined enumeration type.

`C5108 Identifier <name> not declared.`

A reference was made to a the variable <name> which was not declared.

`C5109 Symbol <name> in use for non-function.`

The identifier used in the declaration of a function was already used for something else.

`C5110 Function <name> already defined.`

A function was defined more than once.

`C5111 Function <name> declared with different type.`

The type of a function in a declaration did not match with the type specified in an earlier definition or declaration.

`C5112 Component name <name> exists.`

A component name was used more than once in a structure/union definition.

`C5113 Tag <name> is not a structure cq union tag.`

An identifier used after `struct` was defined as a union tag or vice versa.

`C5114 Structure/union tag <name> already defined.`

A tag in a structure / union definition was defined before.

`C5115 Unresolved reference to label <name>.`

The label was used in a `goto` statement, but was not defined in the same function.

`C5116 Label <name> already defined.`

The label was defined more than once in a function.

`C5117 Redefinition of global <name>.`

The global variable was defined (i.e. space allocated to it) more than once.

Errors in structs, unions and enums

`C6101 Constant expression required for enum constant.`

An explicit definition of an enumeration constant was made with a non constant expression.

`C6102 Integer value required for enum constant.`

An explicit definition of an enumeration constant was made with a non integer value.

```
C6103 Union can not contain bit fields.
C6104 Constant expression required for bit field width.
C6105 Integer value required for bit field width.
C6106 Bit fields are not supported yet.
```

These messages relate to bit fields, which are not yet supported.

`C6107 Syntax error in enum type definition. '}' expected.`

An enumeration list consists of a list of identifiers, separated by commas and enclosed in braces.

```
C6108 Syntax error in enum list. identifier expected as
enum constant.
```

A comma in an enumeration list should always be followed by an identifier.

```
C6109 Syntax error in enum type definition. Tag or '{'
expected.
```

`enum` should be followed by either an identifier, used as enumeration tag, or opening brace, starting the enumeration list.

```
C6110 Missing type specifier in component declaration.
Assumed int.
```

The type specifier in a component declaration was missing.

`C6111 Declarator or bit field expected in component.`

The type specifier in a component declaration was not followed by a declarator.

```
C6112 Syntax error in component declarator list. ',' or
';' expected.
```

The identifiers following the typespecifier in a component declaration should be separated by a comma; declarations should be separated by a semicolon.

```
C6113 Syntax error in struct/union type specifier. Tag or
'{' expected.
```

`struct` or `union` should be followed by an identifier (the structure or union tag) or an opening brace.

```
C6114 Syntax error in struct/union type definition. '}'
expected.
```

A structure/union definition was not terminated by a closing brace.

Compiler warning messages

Warning numbers are given a severity level, which corresponds to the warning suppression level specified by the `-wl` option (see the chapter on the Par.C System). This level is given in parenthesis following the message text.

Declaration assumption

`C7101 Declaration expected. (6)`

`C7102 Useless declaration (no side effects). (6)`

`C7103 Type and storage class specifier missing. (5)`

`C7104 Function <name> declared implicitly. (2)`

The function < name > was declared when it was used for the first time, not by explicit declaration.

Warning about name

`C7201 Assumed function <name> returns int. (2)`

The function definition did not specify the function return type, so `int` was assumed by default.

`C7202 Assumed parameter <name> has type int. (2)`

The formal parameter was not declared. Type integer was assumed by default.

`C7203 Macro <name> redefined. (1)`

The macro < name > was redefined without a preceding `#undef`.

```
C7204 Reserved keyword <name>; possible future
incompatibility. (1)
```

The identifier < name > might in future versions of the Par.C System be used as keyword. Until then it can be used as identifier, but upward compatibility is not guaranteed.

Expressions

`C7301 Illegal character in character constant. (6)`

An illegal character was encountered in a character constant.

`C7302 Illegal escape code in character constant. (6)`

A non valid escape code was encountered in a character constant.

`C7303 Character constant too long. (6)`

A character constant contained more than one character.

`C7304 Integer constant too large for unsigned long. (6)`
`C7306 Decimal constant too large. (6)`

A constant was encountered which exceeded the available range.

`C7307 Divide by zero in constant expression. (6)`

A constant expression could not be evaluated because it required a division by zero.

`C7308 Division/remainder of long constants not implemented. (6)`

A constant expression involving long integers could not be evaluated at compile time.

`C7309 ptr/int conversion. (4)`

A pointer value was converted to an integer before operating on it.

`C7310 int/ptr conversion. (4)`

An integer value was converted to a pointer before using it.

`C7311 ptr/ptr conversion. (3)`

A pointer value was converted to a pointer to another type before using it.

`C7312 Untyped chan-chan assignment; int assumed. (2)`

If both the left side and the right side of an assignment expression are of `channel` type, the compiler cannot determine the number of bytes to communicate. Type integer (four bytes) is assumed.

`C7313 Implicit conversion to void type. (2)`

A value was returned from a function defined with type `void`.

`C7314 Overflow in constant conversion. (6)`

Conversion of a constant to the appropriate type caused an overflow.

Preprocessor

`C7401 End of file reached with unbalanced <name>. (5)`

The preprocessor directive <name> was used without the balancing counterpart.

`C7402 End of file reached with unbalanced #asm. (5)`

`#asm` was used without balancing `#endasm`.

`C7403 End of file reached with unbalanced comment. (6)`

A comment was opened but never closed.

```
C7405 Spurious text following preprocessor command.
Ignored. (1)
```

A preprocessor command was not followed by a newline.

Miscellaneous

`C7501 Illegal character: $XX. Ignored. (6)`

The preprocessor encounterd an illegal character.

`C7502 Source line too long. Terminated. (6)`

The length of a logical source line exceeded the maximum linelength of the compiler.

`C7503 Channel initialization overridden. (6)`

A global variable of `channel` type was explicitly initialised, overriding the default initialisation. This can cause serious problems, because the transputer requires channels to be initialised to `MinInt`, before communication can take place.

Internal compiler error messages

`C8000 Internal Error: <origin>`

If this error message is printed on the screen we kindly ask you to send us a filled out bug report.

Linker error messages

`Booter load address offset segment length error (!=4)`

The segment defining the booter load address offset has been found to be incorrect. This error should not occur when working with C programs. When programming in assembly this error could occur if a wrong segment number has been used (only code and data are allowed).

`Booter load address segment length error (!=4)`

The segment defining the booter load address has been found to be incorrect. This error should not occur when working with C programs. When programming in assembly this error could occur if a wrong segment number has been used (only code and data are allowed).

`Can't write map file`

The linker could not write the map file. It is most likely that your disk is full, or the file exists and is write-protected.

`Code load address segment length error (!=4)`

The segment defining the code load addresss has been found to be incorrect. This error should not occur when working with C programs. When programming in assembly this error could occur if a wrong segment number has been used (only code and data are allowed).

`Code segment: offset too large`

The space reserved for the storage of a global reference offset is insufficient. More space can be reserved using the assembler `-id` and `-ic` options.

`Duplicate segment`

Each type of segment may occur only once in an object module.

`Duplicate code instruction length`

Indicates an invalid object code format. Should never occur.

`Duplicate data instruction length`

Indicates an invalid object code format. Should never occur.

`Duplicate boot block`

Two blocks containing booter code were found. Possibly both the `a.lib` and the `b.lib` libraries were linked.

`Duplicate booter load address`

Two blocks containing booter code were found. Possibly both the `a.lib` and the `b.lib` libraries were linked.

`Duplicate code load address`

Two blocks containing a code load address specification were found.

`Duplicate entry point`

Two program entry points were found. Possibly both the `a.lib` and the `b.lib` libraries were linked.

`Duplicate memory specification`

Two blocks containing a memory specification were found.

`End of file reached prematurely`

An end of file was encountered while there were still segments open.

`Error closing input file`

The input file could not be closed.

`Error opening input file`

The input file could not be opened.

`Error opening output file`

The output file could not be opened. Possible causes are: disk full, directory full or disk is write-protected.

`Error reading file`
`Error reading indirect file`

The file could not be read. Possibly the file is empty, or the disk has a bad sector.

`Error writing output file`

An error occured during the writing of the output file (possibly the disk is full).

```
File error (illegal code label segment)
File error (illegal data label type)
File error (illegal data label segment)
```

These errors indicate an error in the object file format. Should normally not occur.

```
Filename in indirect file too large
```

The filename found in the specified indirect file was too long.

```
Format error reading file
```

An attempt was made to link a file which was not in the object file or library file format.

```
Illegal option
```

The option was not recognized as a valid option.

```
Illegal segment
Illegal tag
```

The object file format is incorrect. Possibly the file has been edited before linking. Otherwise the compiler or assembler could have generated an error in writing their output. In that case please send in a bug report.

```
Label type conflict
```

The label is already in use for another type of object. You can not use the same name for both a variable and a function.

```
Memory segment length error
```

The segment defining the memory specification has been found to be incorrect. This error should not occur when working with C programs. When programming in assembly this error could occur if a wrong segment number has been used (only code and data are allowed).

```
Needs filename
```

The option required a filename but none was found.

```
No booter: can't make stand-alone program!
```

No booter has been found, and therefore no executable program could be produced. This error may occur when not linking either `a.lib` or `b.lib`. This message is not generated when the `-c` option is used.

`No object files`

There were no object modules specified in the commandline.

`Not an object file or library`

The indicated file does not appear to be a library or an object module.

`Option argument too small. minimum taken.`

An option has been used with too small an argument. The minimum value for the argument has been substituted.

`Option argument too large. maximum taken.`

An option has been used with too large an argument. The argument given has been truncated to the maximum size.

`Out of memory`

There was insufficient memory available for the completion of the link process. When working on an IBM-PC with `parcl.exe`, try to remove resident programs, RAM chache and RAM disk.

`Segment error: entry point (segment length != 4)`

The segment defining the program entry point has been found to be incorrect. This error should not occur when working with C programs. When programming in assembly this error could occur if a wrong segment number has been used (only code and data are allowed).

`Symbol multiple defined`

A symbol definition has occured more than once. The linker can not resolve this symbol unambiguously.

`Undefined symbol`

A reference has been found to a symbol that is not defined in the given list of object modules and libraries. It is most likely that you misspelled the symbol name. Remember that the linker is case sensitive. Check whether all the necessary libraries and object files are listed in the command.

Linker warning messages

```
*WARNING* data label XXXX at odd boundary
```

The compiler always locates static data storage on a (32 bit) word address. This warning could show up when inline assembly code has been used.

```
Booter data segment ignored
```

It is not meaningful to use a datasegment in booter code.

Internal linker error messages

```
Internal error <message>
Linker bug <message>
```

If one of these error messages is printed on the screen we kindly ask you to send us a filled out bug report.

ASCII code table

In the following table, the numerical value of the ASCII coded characters are given in decimal and hexadecimal notation, and the character is printed in the "Char" column.

Dec	Hex	Ctrl	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	^@	NUL	32	20	SP	64	40	@	96	60	`
1	01	^A	SOH	33	21	!	65	41	A	97	61	a
2	02	^B	STX	34	22	"	66	42	B	98	62	b
3	03	^C	ETX	35	23	#	67	43	C	99	63	c
4	04	^D	EOT	36	24	$	68	44	D	100	64	d
5	05	^E	ENQ	37	25	%	69	45	E	101	65	e
6	06	^F	ACK	38	26	&	70	46	F	102	66	f
7	07	^G	BEL	39	27	'	71	47	G	103	67	g
8	08	^H	BS	40	28	(	72	48	H	104	68	h
9	09	^I	HT	41	29	)	73	49	I	105	69	i
10	0A	^J	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	^K	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	^L	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	^M	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	^N	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	^O	SI	47	2F	/	79	4F	O	111	6F	o
16	10	^P	DLE	48	30	0	80	50	P	112	70	p
17	11	^Q	DC1	49	31	1	81	51	Q	113	71	q
18	12	^R	DC2	50	32	2	82	52	R	114	72	r
19	13	^S	DC3	51	33	3	83	53	S	115	73	s
20	14	^T	DC4	52	34	4	84	54	T	116	74	t
21	15	^U	NAK	53	35	5	85	55	U	117	75	u
22	16	^V	SYN	54	36	6	86	56	V	118	76	v
23	17	^W	ETB	55	37	7	87	57	W	119	77	w
24	18	^X	CAN	56	38	8	88	58	X	120	78	x
25	19	^Y	EM	57	39	9	89	59	Y	121	79	y
26	1A	^Z	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	^[	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	^\	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	^]	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	^^	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	^_	US	63	3F	?	95	5F	_	127	7F	DEL

References

ANSI [1986]

Draft Proposed American National Standard for Information Systems - Programming Language C

Accredited Standards Committee X3, (ANSI) 1986

HARBISON [1984]

C: a reference manual

Samual P. Harbison & Guy L. Steele Jr.

Prentice-Hall, 1984

INMOS [1986]

Transputer reference manual

Inmos Ltd, 1986

KERNIGHAN [1978]

The C programming language

Brian W. Kernighan & Dennis M. Ritchie

Prentice-Hall, 1978

Index

BUG REPORT FORM

This program has been designed carefully. However, it is always possible that some irregularities have slipped through. If you come across any bug or error in this Par.C System, please fill in a copy of this form and send it to Parsec Developments, P.O.Box 782, 2300 AT Leiden, The Netherlands, Fax +31 71 134449

User information ..	Name:	
	Address:	
	City:	Country:
	Phone number:	
	Par.C Version:	Serial number:
Your Par.C dealer .	Name:	
	Address:	
	City:	Country:

Where? .... Circle the part of the Par.C System in which you located an error ?

Compiler, assembler, linker, loader/server, runtime functions.

Manual: page:

Other:

Hardware .... **transputer network** (layout, types, processorspeeds, amount of memory)...

host system (type, operating system, memory, I/O recources, system settings), link interface, etc.

Please turn over

Error description . Which command line options were used?...

Which error message(s) were generated?...

Which standard libraries were linked to the program?...

Is the error reproducable?...

If possible, please include a printout of the source text(s) which caused generation of the error. It will be appreciated if you remove all irrelevant information. Please include a description of what the code was supposed to do and what it actually did.