LIBS

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
799 - laser1 Sat Feb 9 19:41:33 EST 1985

ARGS: [-C -d -fp -oorig=c++ -otime=476843667]
NAME: [c++]
PASSCOMM: [sfbc!c++:D.Swartout:lpbin=m13:home=bc]
copies: [1]
ctime: [7]
files: [/usr/spool/lp/request/laser/d0-9681]
job: [laser-9005]
lpbin: []
mach: [sfbc (laser-9005)]
mail: []
originator: [c++]
printer: [laser1]
printtime: [476843932]
request: [laser-9681]
route: [nsc]
sequence: [799]
subtime: [476843667]
title: []
transit: [4]
version: [08-02-84]

```
# %Z% %M% %I% %H% %T%
#
#       Lib master makefile.
#
CC=CC
OSUF=.o

all:
        cd complex ; make CC=$(CC) OSUF=$(OSUF)
        cd stream ; make CC=$(CC) OSUF=$(OSUF)
        cd task ; make CC=$(CC) OSUF=$(OSUF)
        cd new ; make CC=$(CC) OSUF=$(OSUF)
        ar cr libC.a */*$(OSUF)

clean:
        cd complex ; make clean
        cd stream ; make clean
        cd task ; make clean
        cd new ; make clean

clobber:        clean
        rm -f libC.a
```

```
# %Z% %M% %I% %H% %T%
CC=CC
CFLAGS=-c
OSUF=.o
OBJS=abs$(OSUF) arg$(OSUF) cos$(OSUF) error$(OSUF) exp$(OSUF) io$(OSUF) \
        log$(OSUF) oper$(OSUF) polar$(OSUF) pow$(OSUF) sin$(OSUF) sqrt$(OSUF)
HDRS=../../incl/complex.h const.h
all     :       $(OBJS)

abs$(OSUF)      :       $(HDRS) abs.c
                $(CC) $(CFLAGS) abs.c

arg$(OSUF)      :       $(HDRS) arg.c
                $(CC) $(CFLAGS) arg.c

cos$(OSUF)      :       $(HDRS) cos.c
                $(CC) $(CFLAGS) cos.c

error$(OSUF)    :       $(HDRS) error.c
                $(CC) $(CFLAGS) error.c

exp$(OSUF)      :       $(HDRS) exp.c
                $(CC) $(CFLAGS) exp.c

io$(OSUF)       :       $(HDRS) io.c
                $(CC) $(CFLAGS) io.c

log$(OSUF)      :       $(HDRS) log.c
                $(CC) $(CFLAGS) log.c

oper$(OSUF)     :       $(HDRS) oper.c
                $(CC) $(CFLAGS) oper.c

polar$(OSUF)    :       $(HDRS) polar.c
                $(CC) $(CFLAGS) polar.c

pow$(OSUF)      :       $(HDRS) pow.c
                $(CC) $(CFLAGS) pow.c

sin$(OSUF)      :       $(HDRS) sin.c
                $(CC) $(CFLAGS) sin.c

sqrt$(OSUF)     :       $(HDRS) sqrt.c
                $(CC) $(CFLAGS) sqrt.c

clean   :
                rm -f $(OBJS) *.i *..c

clobber :       clean
```

```
/* %Z% %M% %I% %H% %T% */

#include "../../incl/complex.h"
#include "const.h"

double abs(complex z)
{
        register double temp;
        register double r = z.re;
        register double i = z.im;

        if (r < 0) r = -r;
        if (i == 0) return r;

        if (i < 0) i = -i;
        if (r == 0) return i;

        if (i > r) { temp = r; r = i; i = temp; }
        temp = i/r;
        temp = r*sqrt(1.0 + temp*temp);  /*overflow!!*/
        return temp;
}

double norm(complex z)
{
        register double r = z.re;
        register double i = z.im;

#define SAFE 6.521908912666391000e+18  /* 0.5*sqrt(GREATEST)*/

        if (r<SAFE && i<SAFE) return r*r+i*i;
        return GREATEST;

}
```

```
/* %Z% %M% %I% %H% %T% */

#include "../../incl/complex.h"

double  arg(complex z)
{
        return atan2(z.im,z.re);
}
```

```
/* %Z% %M% %I% %H% %T% */

#include "../../incl/complex.h"
#include "const.h"

complex cos(complex z)
/*
        The cosine of z: cos(z)=cosh(iz).
*/
{
        complex y(-z.im, z.re);
        return cosh(y);
}



complex cosh(complex z)
/*
        The complex hyperbolic cosine of z
*/
{
        double  cosh_x, sinh_x, cos_y, sin_y;

#define COSH_GOOD       1e7
#define COSH_HUGE       1e38

        if (z.re > MAX_EXPONENT) {
                complex_error(C_COSH_RE, z.re);
                cosh_x = sinh_x = COSH_HUGE;
        }
        else if (z.re < MIN_EXPONENT) {
                complex_error(C_COSH_RE, z.re);
                cosh_x = COSH_HUGE;
                sinh_x = -COSH_HUGE;
        }
        else {
                double pos_exp = exp(z.re);
                double neg_exp = 1/pos_exp;
                cosh_x = (pos_exp + neg_exp)/2;
                sinh_x = (pos_exp - neg_exp)/2;
        }

        if (ABS(z.im) > COSH_GOOD) {
                complex_error(C_COSH_IM, z.im);
                cos_y = sin_y = 0;
        }
        else {
                cos_y = cos(z.im);
                sin_y = sin(z.im);
        }

        return complex(cos_y*cosh_x, sin_y*sinh_x);
}
```

```
/* %Z% %M% %I% %H% %T% */
#include "../../incl/complex.h"

void complex_error(int err, double)
{
        errno = err;
}
```

```
/* %Z% %M% %I% %H% %T% */

#include "../../incl/complex.h"
#include "const.h"


complex exp(complex z)
/*
        The complex exponentiation function: e**z, e being 2.718281828...

        In case of overflow, return EXPHUGE with the appropriate phase.
        In case of underflow return 0.
        In case of ridiculous input to "sin" and "cos", return 0.
*/
{
        complex answer;
        double  radius, sin_theta, cos_theta;

#define EXPHUGE 1e38
#define EXPGOOD 1e7

        if (z.re > MAX_EXPONENT) {
                complex_error(C_EXP_RE_POS, z.re);
                radius = EXPHUGE;
        }
        else if (z.re < MIN_EXPONENT) {
                complex_error(C_EXP_RE_NEG, z.re);
                radius = 0;
        }
        else {
                radius = exp(z.re);
        }

        if (z.im > EXPGOOD || z.im < -EXPGOOD) {
                complex_error(C_EXP_IM, z.im);
                sin_theta = cos_theta = 0;
        }
        else {
                sin_theta = sin(z.im);
                cos_theta = cos(z.im);
        }

        answer.re = radius * cos_theta;
        answer.im = radius * sin_theta;

        return answer;
}
```

```
/* %Z% %M% %I% %H% %T% */
#include "../../incl/complex.h"

ostream& operator<<(ostream& s, complex a)
{
        return s<<"( "<<real(a)<<", "<<imag(a)<<")";
}

istream& operator>>(istream& s, complex& a)
/*
        f
        ( f )
        ( f , f )
*/
{
        double  re = 0, im = 0;
        char    c = 0;

        s>>c;
        if (c == '(') {
                s>>re>>c;
                if (c == ',') s>>im>>c;
                if (c != ')') s.clear(_bad);
        }
        else {
                s.putback(c);
                s>>re;
        }

        if (s) a = complex(re,im);
        return s;
}
```

```
/* %Z% %M% %I% %H% %T% */

#include "../../incl/complex.h"
#include "const.h"

#define LOGWILD -1000
#define LOGDANGER       1e18
#define PERIL(t) (t > LOGDANGER || (t < 1/LOGDANGER && t != 0) )

complex log(complex z)
/*
        The complex natural logarithm of "z".
        If z = 0, then the answer LOGWILD + 0*i is returned.

        Stu Feldman says that the peril tests for the following function
        are "acceptable for now", but certain things like
        complex variables outside the over/underflow range
        will cause floating exceptions.
*/
{
        complex answer;
        double  partial;

        if ( z.re == 0 && z.im == 0) {
                complex_error(C_LOG_0,0);
                answer.re = LOGWILD;
        }
        else {
        /*
                 Check for (over/under)flow, and fixup if necessary.
        */
                double x = ABS(z.re);
                double y = ABS(z.im);

                if ( x>y && PERIL(x) ) {
                        z.im /=x;
                        z.re /= x;  /* z.re is replaced by 1 or -1 */
                        partial = log(x);
                }
                else if (PERIL(y)) {
                        z.im /= y;  /* roles of re, im reversed from previous */
                        z.re /= y;
                        partial = log(y);
                }
                else partial = 0;

        /*
                z.re*z.re and z.im*z.im should not cause problems now.
        */

                answer.im = atan2(z.im,z.re);
                answer.re = log(z.re*z.re + z.im*z.im)/2 + partial;
        }
        return answer;
}
```

```
/* %Z% %M% %I% %H% %T% */

#include "../../incl/complex.h"
#include "const.h"

complex operator*(complex a1,complex a2)
{
        return complex(a1.re*a2.re-a1.im*a2.im, a1.re*a2.im+a1.im*a2.re);
}


complex operator/(complex a1, complex a2)
{
        register double r = a2.re;      /* (r,i) */
        register double i = a2.im;
        register double ti;             /* (tr,ti) */
        register double tr;

        tr = ABS(r);
        ti = ABS(i);

        if (tr <= ti) {
                ti = r/i;
                tr = i * (1 + ti*ti);
                r = a1.re;
                i = a1.im;
        }
        else {
                ti = -i/r;
                tr = r * (1 + ti*ti);
                r = -a1.im;
                i = a1.re;
        }

        return complex( (r*ti + i)/tr, (i*ti - r)/tr );
}

void complex.operator*=(complex a)
{
        register double r = re*a.re - im*a.im;
        register double i = re*a.im + im*a.re;
        re = r;
        im = i;
}

void complex.operator/=(complex a)
{
        complex quot, temp1, temp2;

        if ( (temp2.re = a.re) < 0 ) temp2.re = -temp2.re;
        if ( (temp2.im = a.im) < 0 ) temp2.im = -temp2.im;
        if ( temp2.re <= temp2.im) {
                temp2.im = a.re/a.im;
                temp2.re = a.im * (1 + temp2.im*temp2.im);
                temp1 = *this;
        }
```

```
        else {
                temp2.im = -a.im/a.re;
                temp2.re = a.re * (1 + temp2.im*temp2.im);
                temp1.re = -im;
                temp1.im = re;
        }
        re = (temp1.re * temp2.im + temp1.im) / temp2.re;
        im = (temp1.im * temp2.im - temp1.re) / temp2.re;
}
```

```
/* %Z% %M% %I% %H% %T% */

#include        "../../incl/complex.h"

complex polar(double r, double theta)
{
        return complex(r * cos(theta), r * sin(theta) );
}
```

```
/* %Z% %M% %I% %H% %T% */

#include "../../incl/complex.h"

complex pow(double base, complex z)
/*
        real to complex power: base**z.
*/
{
        register complex y;

        if (base == 0) return y;        /* even for singularity */

        if (0 < base) {
                double lb = log(base);
                y.re = z.re * lb;
                y.im = z.im * lb;
                return exp(y);
        }

        return pow(complex(base), z);   /* use complex power fct */
}


complex  pow(complex a, int n)
/*
        complex to integer power: a**n.
*/
{
        register complex x, p = 1;

        if (n == 0) return p;

        if (n < 0) {
                n = -n;
                x = 1/a;
        }
        else    x = a;

        for( ; ; ) {
                if(n & 01) {
                        register double t = p.re * x.re - p.im * x.im;
                        p.im = p.re * x.im + p.im * x.re;
                        p.re = t;
                }
                if(n >>= 1) {
                        register double t = x.re * x.re - x.im * x.im;
                        x.im = 2 * x.re * x.im;
                        x.re = t;
                }
                else    break;
        }
        return p;
}

complex pow(complex a, double b)
```

```
/*
        complex to real power: a**b.
*/
{
        register double logr = log( abs(a) );
        register double logi = atan2(a.im, a.re);
        register double x = exp( b*logr );
        register double y = b * logi;
        return complex(x*cos(y), x*sin(y));
}


complex pow(complex base, complex sup)
/*
        complex to complex power: base**sup.
*/
{
        complex result;
        register double logr, logi;
        register double xx, yy;
        double a = abs(base);

        if (a == 0) return result;

        logr = log( a );
        logi = atan2(base.im, base.re);

        xx = exp( logr * sup.re - logi * sup.im );
        yy = logr * sup.im + logi * sup.re;

        result.re = xx * cos(yy);
        result.im = xx * sin(yy);

        return result;
}
```

```
/* %Z% %M% %I% %H% %T% */

#include "../../incl/complex.h"
#include "const.h"

complex sin(complex z)
/*
        sine of z: -i * sinh(i*z)
*/
{
        complex y = complex(-z.im, z.re);       /*  i * z */
        y = sinh(y);                            /*  csinh(y) */
        return complex(y.im, -y.re);            /*  -i * y */
}

complex sinh(complex z)
/*
        The hyperbolic sine
*/
{
        double cosh_x, sinh_x, cos_y, sin_y;

#define SINH_GOOD       1e7
#define SINH_HUGE       1e38

        if (z.re > MAX_EXPONENT) {
                complex_error(C_SINH_RE, z.re);
                cosh_x = sinh_x = SINH_HUGE;
        }
        else if (z.re < MIN_EXPONENT) {
                complex_error(C_SINH_RE, z.re);
                cosh_x = SINH_HUGE;
                sinh_x = -SINH_HUGE;
        }
        else {
                double pos_exp = exp(z.re);
                double neg_exp = 1/pos_exp;
                cosh_x = (pos_exp + neg_exp)/2;
                sinh_x = (pos_exp - neg_exp)/2;
        }

        if (ABS(z.im) > SINH_GOOD) {
                complex_error(C_SINH_IM, z.im);
                cos_y = sin_y = 0;
        }
        else  {
                cos_y = cos(z.im);
                sin_y = sin(z.im);
        }

        return complex(cos_y*sinh_x, sin_y*cosh_x);
}
```

```
/* %Z% %M% %I% %H% %T% */
# include "../../incl/complex.h"


#define SQRT_DANGER     1e17
# define PERIL(t)       (t > SQRT_DANGER || (t < 1/SQRT_DANGER && t != 0))

/*
 *
 * 7-25-83, note from Leonie Rose -
 * Stu Feldman says that the peril tests for the following function
 * are "acceptable" for now, but certain things like
 * sqrt(1e10 + 1e-30*i) will cause floating exceptions.
 *
 */

complex sqrt(complex z)
{
        complex  answer;
        double  r_old,  partial;

/*
        Check for possible overflow, and fixup if necessary.
*/

        double x = abs(z.re);
        double y = abs(z.im);

        if (x > y && PERIL(x)) {
                z.im /= x;
                z.re /= x;  /* z.re is replaced by 1 or -1 */
                partial = sqrt(x);
        }
        else if PERIL(y) {
                z.im /= y;  /* roles of z.re, z.im reversed from previous */
                z.re /= y;
                partial = sqrt(y);
        }
        else partial = 1;

/*
        Main computation:
        Use half angle formulas to compute angular part of the square root.
        The sign of sin_old is the same as that for sin_new, which means that the
        upper half plane gets mapped to the first quadrant, and
        the lower half plane to the fourth quandrant.
*/


        if (r_old = sqrt(z.re*z.re + z.im*z.im)) {
                double r_new = partial * sqrt(r_old);

                double cos_old = z.re/r_old;
                double sin_old = z.im/r_old;
                double cos_new = sqrt( (1 + cos_old)/2 );
                double sin_new = (cos_new == 0)? 1 : sin_old/(2*cos_new);
```

```
                answer.re = r_new * cos_new;
                answer.im = r_new * sin_new;
        }

        return answer;
}
```

```
# %Z% %M% %I% %H% %T%
CC=CC
CFLAGS=-c
OSUF=.o
OBJS=_delete$(OSUF) _handler$(OSUF) _new$(OSUF) _vec$(OSUF)
all     :       $(OBJS)

_delete$(OSUF)  :       _delete.c
                $(CC) $(CFLAGS) _delete.c

_handler$(OSUF) :       _handler.c
                $(CC) $(CFLAGS) _handler.c

_new$(OSUF)     :       _new.c
                $(CC) $(CFLAGS) _new.c

_vec$(OSUF)     :       _vec.c
                $(CC) $(CFLAGS) _vec.c

clean   :
                rm -f $(OBJS) *.i *..c

clobber :       clean
```

```
/* %Z% %M% %I% %H% %T% */
free(char*);
extern void operator delete(void* p)
{
        if (p) free( (char*)p );
}
```

```
/* %Z% %M% %I% %H% %T% */

typedef void (*PFVV)();

extern PFVV _new_handler = 0;

extern PFVV set_new_handler(PFVV handler)
{
        PFVV rr = _new_handler;
        _new_handler = handler;
        return rr;
}
```

```
/* %Z% %M% %I% %H% %T% */

typedef void (*PFVV)();

extern PFVV _new_handler;

extern void* operator new(/* long !!!!!! */ unsigned size)
{
        extern char* malloc(unsigned);
        char* p;

        while ( (p=malloc(size))==0 ) {
                if(_new_handler)
                        (*_new_handler)();
                else
                        return 0;
        }
        return (void*)p;
}
```

```
/* %Z% %M% %I% %H% %T% */
typedef void* PV;
typedef void (*PF)(PV);

extern PV _vec_new(PV op, int n, int sz, PV f)
/*
        allocate a vector of "n" elements of size "sz"
        and initialize each by a call of "f"
*/
{
        register int i;
        register char* p;
        if (op == 0) op = PV( new char[n*sz] );
        p = (char*) op;
        for (i=0; i<n; i++) ( *PF(f) )( PV(p+i*sz) );
        return PV(p);
}

void _vec_delete(PV op, int n, int sz, PV f)
{
        register int i;
        register char* p = (char*) op;
        for (i=0; i<n; i++) ( *(PF)f )( (PV)(p+i*sz) );
}
```

```
# %Z% %M% %I% %H% %T%
CC=CC
CFLAGS=-c
OSUF=.o
OBJS=circbuf$(OSUF) filebuf$(OSUF) in$(OSUF) out$(OSUF) streambuf$(OSUF)
HDRS=../../incl/stream.h
all     :       $(OBJS)

circbuf$(OSUF)  :       $(HDRS) circbuf.c
                $(CC) $(CFLAGS) circbuf.c

filebuf$(OSUF)  :       $(HDRS) filebuf.c
                $(CC) $(CFLAGS) filebuf.c

in$(OSUF)       :       $(HDRS) in.c
                $(CC) $(CFLAGS) in.c

out$(OSUF)      :       $(HDRS) out.c
                $(CC) $(CFLAGS) out.c

streambuf$(OSUF)        :       $(HDRS) streambuf.c
                $(CC) $(CFLAGS) streambuf.c

clean   :
                rm -f $(OBJS) *.i *..c

clobber :       clean
```

```
/* %Z% %M% %I% %H% %T% */
#include "../../incl/stream.h"

/*
 *      Come here on a put to a full buffer.  Allocate the buffer if
 *      it is uninitialized.
 *      Returns:        EOF on error
 *                      the input character on success
 */
virtual int circbuf.overflow(char c)
{
        if (allocate() == EOF) return EOF;

        pptr = base;
        if (c != EOF) *pptr++ = c;

        return c & 0377;
}

/*
 *      Fill a buffer.
 *      Returns:        EOF on error or end of input
 *                      next character on success
 */
virtual int circbuf.underflow()
{
        return EOF;
}
```

```
/* %Z% %M% %I% %H% %T% */
#include "../../incl/stream.h"

/* define some UNIX calls */
extern int open (char *, int);
extern int close (int);
extern long lseek (int, long, int);
extern int read (int, char *, unsigned);
extern int write (int, char *, unsigned);
extern int creat (char *, int);

/*
 *      Open a file with the given mode.
 *      Return:         NULL if failure
 *                      this if success
 */
filebuf* filebuf.open (char *name, open_mode om)
{
        switch (om) {
        case input:
                fd = ::open(name, 0);
                break;
        case output:
                fd = ::open(name, 1);
                break;
        case append:
                if ((fd = ::open(name, 1)) >= -1) {
                        if (lseek(fd, 0, 2) < 0) {
                                (void)::close(fd);
                                fd = -1;
                        }
                } else
                        fd = creat(name, 664);
                break;
        }
        if (fd < 0)
                return NULL;
        else {
                opened = 1;
                return this;
        }
}

/*
 *      Empty an output buffer.
 *      Returns:        EOF on error
 *                      0 on success
 */
int filebuf.overflow(int c)
{
        if (!opened)
                return EOF;

        if (allocate() == EOF)
                        return EOF;
```

```
        if (pptr > base)
                if (write(fd, base, pptr-base) != pptr-base)
                        return EOF;

        pptr = gptr = base;
        if (c != EOF)
                *pptr++ = c;
        return c & 0377;
}

/*
 *      Fill an input buffer.
 *      Returns:        EOF on error or end of input
 *                      next character on success
 */
int filebuf.underflow()
{
        int count;

        if (!opened) return EOF;

        if (allocate() == EOF) return EOF;

        if ((count=read(fd, base, eptr - base)) < 1) return EOF;

        gptr = base;
        pptr = base + count;
        return *gptr & 0377;
}
```

```
/* %Z% %M% %I% %H% %T% */
/*
        C++ stream i/o source

        in.c
*/
#include <ctype.h>
/*#include <stdio.h>*/
#include "../../incl/stream.h"
#include "../../incl/common.h"

/* the predefined streams */
filebuf cin_file = {
        /*base stuff*/0, 0, 0, 0, 0, filebuf__vtbl, /*fd*/0, /*opened*/1
};
istream cin = { /*bp*/(streambuf*)&cin_file, /* tied_to */&cout, /*skipws*/1, /*stat

/* predefined whitespace */
whitespace WS;

/*inline */void eatwhite (istream& is)
{
        if (is.tied_to) is.tied_to->flush();
        register streambuf *nbp = is.bp;
        register char c = nbp->sgetc();
        while (isspace(c)) c = nbp->snextc();
}

istream& istream.operator>>(whitespace& w)
{
        register char c;
        register streambuf *nbp = bp;

        &w;
        if (state) return *this;
        if (tied_to) tied_to->flush();
        c = nbp->sgetc();
        while (isspace(c)) c = nbp->snextc();

        if (c == EOF) state |= _eof;

        return *this;
}

istream& istream.operator>>(register char& s)
/*
        reads characters NOT very small integers
 */
{
        if (skipws) eatwhite(*this);

        if (state) {
                state |= _fail;
                return *this;
        }
```

```
        s = bp->sgetc();
        if (s == EOF) {
                state |= _fail|_eof;
                return *this;
        }

        if (bp->snextc() == EOF) state |= _eof;
        return *this;
}

istream& istream.operator>>(register char* s)
{
        register c;
        register streambuf *nbp = bp;

        if (skipws) eatwhite(*this);

        if (state) {
                state |= _fail;
                return *this;
        }

        /* get string */
        c = nbp->sgetc();
        if (c == EOF)
                state |= _fail;
        while (!isspace(c) && c != EOF) {
                *s++ = c;
                c = nbp->snextc();
        }
        *s = '\0';

        if (c == EOF) state |= _eof;

        return *this;
}

istream&
istream.operator>>(long& i)
{
        register c;
        register ii = 0;
        register streambuf *nbp = bp;
        int neg = 0;

        if (skipws) eatwhite(*this);

        if (state) {
                state |= _fail;
                return *this;
        }

        switch (c = nbp->sgetc()) {
        case '-':
        case '+':
                neg = c;
```

```
                c = nbp->snextc();
                break;
        case EOF:
                state |= _fail;
        }

        if (isdigit(c)) {
                do {
                        ii = ii*10+c-'0';
                } while (isdigit(c=nbp->snextc()));
                i = (neg=='-') ? -ii : ii;
        } else
                state |= _fail;

        if (c == EOF) state |= _eof;
        return *this;
}

istream&
istream.operator>>(int& i)
{
        long l;

        if (skipws) eatwhite(*this);

        if (state) {
                state |= _fail;
                return *this;
        }

        if ( *this>>l ) {
                i = l;
        }
        return *this;
}

istream&
istream.operator>>(short& i)
{
        long l;

        if (skipws) eatwhite(*this);

        if (state) {
                state |= _fail;
                return *this;
        }

        if ( *this>>l ) {
                i = l;
        }
        return *this;
}

istream&
istream.operator>>(double& d)
```

```
/*
        {+|-} d* {.} d* { e|E {+|-} d+ }
        except that
                - a dot must be pre- or succeded by at least one digit
                - an exponent must be preseded by at least one digit
*/
{
        register c = 0;
        char buf[256];
        register char* p = buf;
        register streambuf* nbp = bp;
        extern double atof(char*);

        if (skipws) eatwhite(*this);

        if (state) {
                state |= _fail;
                return *this;
        }

        /* get the sign */
        switch (c = nbp->sgetc()) {
        case EOF:
                state = _eof|_fail;
                return *this;
        case '-':
        case '+':
                *p++ = c;
                c = bp->snextc();
        }

        /* get integral part */
        while (isdigit(c)) {
                *p++ = c;
                c = bp->snextc();
        }

        /* get fraction */
        if (c == '.') {
                do {
                        *p++ = c;
                        c = bp->snextc();
                } while (isdigit(c));
        }

        /* get exponent */
        if (c == 'e' || c == 'E') {
                *p++ = c;
                switch (c = nbp->snextc()) {
                case EOF:
                        state = _eof|_fail;
                        return *this;
                case '-':
                case '+':
                        *p++ = c;
                        c = bp->snextc();
```

```
                }
                while (isdigit(c)) {
                        *p++ = c;
                        c = bp->snextc();
                }
        }

        *p = 0;
        d = atof(buf);

        if (c == EOF) state |= _eof;
        return *this;
}

istream&
istream.operator>>(float& f)
{
        double d;

        if (skipws) eatwhite(*this);

        if (state) {
                state |= _fail;
                return *this;
        }

        if ( *this>>d ) {
                f = d;
        }
        return *this;
}

istream&
istream.get(
        register char* s,       /* character array to read into */
        register int len,       /* size of character array */
        register char term      /* character that terminates input */
) {
        register c;
        register streambuf *nbp = bp;

        if (state) {
                state |= _fail;
                return *this;
        }

        if ((c = bp->sgetc()) == EOF) {
                state |= _fail | _eof;
                return *this;
        }

        while (c != term && c != EOF && len > 1) {
                *s++ = c;
                c = nbp->snextc();
                len--;
        }
```

```
        *s = '\0';
        if (c == EOF)
                state |= _eof;
        return *this;
}

istream&
istream.putback(        register char c /* character to put back */) {
        bp->sputbackc(c);
        return *this;
}


istream&
istream.get(
        register streambuf &s,  /* streambuf to input to */
        register char term      /* termination character */
){
        register c;
        register streambuf *nbp = bp;

        if (state) {
                state |= _fail;
                return *this;
        }

        if ((c = bp->sgetc()) == EOF) {
                state |= _fail | _eof;
                return *this;
        }

        while (c != term && c != EOF) {
                if (s.sputc(c) == EOF)
                        break;
                c = nbp->snextc();
        }
        if (c == EOF)
                state |= _eof;
        return *this;
}

istream&
istream.operator>>(register streambuf &s) {
        register c;
        register streambuf *nbp = bp;

        if (state) {
                state |= _fail;
                return *this;
        }

        if ((c = bp->sgetc()) == EOF) {
                state |= _fail | _eof;
                return *this;
        }
```

```
        while (c != EOF) {
                if (s.sputc(c) == EOF)
                        break;
                c = nbp->snextc();
        }
        if (c == EOF)
                state |= _eof;
        return *this;
}

istream& istream.operator>>(common& p)
{
        return p.read(*this);
}
```

```
/* %Z% %M% %I% %H% %T% */
/*
        C++ stream i/o source

        out.c
*/
/*#include <stdio.h>*/
sprintf(char*,char* ...);
strlen(char*);
#include "../../incl/stream.h"
#include "../../incl/common.h"


#define MAXOSTREAMS 20

/*
 *      This is a monumental hack which will soon become illegal.  The
 *      initializers depend on the number of elements in the base
 *      type (streambuf), the virtual pointer in the base type, and the
 *      number of elements in the derived type (filebuf).  See cio.h for
 *      their definitions.
 */
filebuf cout_file = {
        0, 0, 0, 0, 0, filebuf__vtbl, 1, 1
};
char cerr_buf[1];
filebuf cerr_file = {
        cerr_buf, cerr_buf, cerr_buf, cerr_buf, 0, filebuf__vtbl, 2, 1
};

ostream cout = { (streambuf*)/*bp*/&cout_file, /*state*/0 };
ostream cerr = { (streambuf*)/*bp*/&cerr_file, /*state*/0 };

const   cb_size = 512;
const   fld_size = 128;

/* a circular formating buffer */
char    formbuf[cb_size];
char*   free=formbuf;
char*   max = &formbuf[cb_size-1];

char* chr(register i, register int w)   /* note: chr(0) is "" */
{
        register char* buf = free;

        if (w<=0 || fld_size<w) w = 1;
        w++;                                    /* space for trailing 0 */
        if (max < buf+w) buf = formbuf;
        free = buf+w;
        char * res = buf;

        w -= 2;                                 /* pad */
        while (w--) *buf++ = ' ';
        if (i<0 || 127<i) i = ' ';
        *buf++ = i;
        *buf = 0;
```

```
        return res;
}

char* str(char* s, register int w)
{
        register char* buf = free;
        int ll = strlen(s);
        if (w<=0 || fld_size<w) w = ll;
        if (w < ll) ll = w;
        w++;                                    /* space for traling 0 */
        if (max < buf+w) buf = formbuf;
        free = buf+w;
        char* res = buf;

        w -= (ll+1);                            /* pad */
        while (w--) *buf++ = ' ';
        while (*s) *buf++ = *s++;
        *buf = 0;
        return res;
}

char* form(char* format ...)
{
        register* ap = (int*)&format;
        register char* buf = free;
        if (max < buf+fld_size) buf = formbuf;

        register ll = sprintf(buf,format,ap[1],ap[2],ap[3],ap[4],ap[5],ap[6],ap[7],
        if (0<ll && ll<1024)                                    /* length */
                ;
        else if (buf<(char*)ll && (char*)ll<buf+1024)   /* pointer to trailing 0 */
                ll = (char*)ll - buf;
        else
                ll = strlen(buf);
        if (fld_size < ll) exit(10);
        free += (ll+1);
        return buf;
}

const char a10 = 'a'-10;

char* hex(long i, register w)
{
        int m = sizeof(long)*2;         /* maximum hex digits for a long */
        if (w<0 || fld_size<w) w = 0;
        int sz = (w?w:m)+1;
        register char* buf = free;
        if (max < buf+sz) buf = formbuf;
        register char* p = buf+sz;
        free = p+1;
        *p-- = 0;                       /* trailing 0 */

        if (w) {
                do {
                        register h = i&0xf;
                        *p-- = (h < 10) ? h+'0' : h+a10;
```

```
                } while (w-- && (i>>=4));
                while (w--) *p-- = ' ';
        }
        else {
                do {
                        register h = i&0xf;
                        *p-- = (h < 10) ? h+'0' : h+a10;
                } while (i>>=4);
        }
        return p+1;
}

char* oct(long i, int w)
{
        int m = sizeof(long)*3;         /* maximum oct digits for a long */
        if (w<0 || fld_size<w) w = 0;
        int sz = (w?w:m)+1;
        register char* buf = free;
        if (max < buf+sz) buf = formbuf;
        register char* p = buf+sz;
        free = p+1;
        *p-- = 0;                       /* trailing 0 */

        if (w) {
                do {
                        register h = i&07;
                        *p-- = h+'0';
                } while (w-- && (i>>=3));
                while (w--) *p-- = ' ';
        }
        else {
                do {
                        register h = i&07;
                        *p-- = h+'0';
                } while (i>>=3);
        }

        return p+1;
}

char* dec(long i, int w)
{
        int m = sizeof(long)*3;         /* maximum dec digits for a long */
        if (w<0 || fld_size<w) w = 0;
        int sz = (w?w:m)+1;
        register char* buf = free;
        if (max < buf+sz) buf = formbuf;
        register char* p = buf+sz;
        free = p+1;
        *p-- = 0;                       /* trailing 0 */

        if (w) {
                do {
                        register h = i%10;
                        *p-- = h + '0';
                } while (w-- && (i/=10));
```

```
                while (w--) *p-- = ' ';
        }
        else {
                do {
                        register h = i%10;
                        *p-- = h + '0';
                } while (i/=10);
        }

        return p+1;
}


ostream& ostream.operator<<(char* s)
{
        register streambuf* nbp = bp;
        if (state) return *this;
        if (*s == 0) return *this;
        do
                if (nbp->sputc(*s++) == EOF) {
                        state |= _eof|_fail;
                        break;
                }
        while (*s);
        if (*(s-1)=='\n') flush(); /* fudge due to lack of  destructors for static*
        return *this;
}

ostream& ostream.operator<<(long i)
{
        register streambuf* nbp = bp;
        register long j;
        char buf[32];
        register char *p = buf;

        if (state) return *this;

        if (i < 0) {
                nbp->sputc('-');
                j = -i;
        } else
                j = i;
        do {
                *p++ = '0' + j%10;
                j = j/10;
        } while (j > 0);
        do {
                if (nbp->sputc(*--p) == EOF) {
                        state |= _fail | _eof;
                        break;
                }
        } while (p != buf);
        return *this;
}

ostream& ostream.put(char c)
```

```
{
        if (state) return *this;

        if (bp->sputc(c) == EOF) state |= _eof|_fail;

        return *this;
}

ostream& ostream.operator<<(double d)
{
        register streambuf* nbp = bp;
        char buf[32];
        register char *p = buf;

        if (state) return *this;

        sprintf(buf,"%g",d);
        while (*p != '\0')
                if (nbp->sputc(*p++) == EOF) {
                        state |= _eof|_fail;
                        break;
                }
        return *this;
}

ostream& ostream.operator<<(streambuf& b)
{
        register streambuf* nbp = bp;
        register int c;

        if (state) return *this;

        c = b.sgetc();
        while (c != EOF) {
                if (nbp->sputc(c) == EOF) {
                        state |= _eof|_fail;
                        break;
                }
                c = b.snextc();
        }

        return *this;
}

/*
 *      empty out an output buffer
 */
ostream& ostream.flush() {
        bp->overflow(EOF);
        return *this;
}


/*
 *      cleanup on exit.
 */
```

```
void _cleanup();

void _Cleanup()
{
        /* flush the stream file buffers */
        cout.flush();
        cerr.flush();

        /* flush stdio */
        _cleanup();
}

void _Exit(int code)
{
        _Cleanup();
        exit(code);
}
```

```
/* %Z% %M% %I% %H% %T% */
#include "../../incl/stream.h"

/*
 *      Allocate some space for the buffer.
 *      Returns:        EOF on error
 *                      0 on success
 */
int streambuf.allocate()
{
        if (base == NULL) {
                if ((base = new char[BUFSIZE]) != NULL) {
                        pptr = gptr = base;
                        eptr = base + BUFSIZE;
                        alloc = 1;
                        return 0;
                } else
                        return EOF;
        }
        return 0;
}

/*
 *      Come here on a put to a full buffer.  Allocate the buffer if
 *      it is uninitialized.
 *      Returns:        EOF on error
 *                      the argument on success
 */
virtual int streambuf.overflow(int c)
{
        if (allocate() == EOF) return EOF;
        if (c != EOF) *pptr++ = c;
        return c&0377;
}

/*
 *      Fill a buffer.
 *      Returns:        EOF on error or end of input
 *                      next character on success
 */
virtual int streambuf.underflow()
{
        return EOF;
}
```

```
# %Z% %M% %I% %H% %T%
CC=CC
CFLAGS=-c
OSUF=.o
OBJS=obj$(OSUF) qhead$(OSUF) qtail$(OSUF) sched$(OSUF) \
        sim$(OSUF) task$(OSUF) timer$(OSUF) swap$(OSUF)
HDRS=../../incl/task.h

all     :       $(OBJS)

obj$(OSUF)      :       $(HDRS) obj.c
                $(CC) $(CFLAGS) obj.c

qhead$(OSUF)    :       $(HDRS) qhead.c
                $(CC) $(CFLAGS) qhead.c

qtail$(OSUF)    :       $(HDRS) qtail.c
                $(CC) $(CFLAGS) qtail.c

sched$(OSUF)    :       $(HDRS) sched.c
                $(CC) $(CFLAGS) sched.c

sim$(OSUF)      :       $(HDRS) sim.c
                $(CC) $(CFLAGS) sim.c

task$(OSUF)     :       $(HDRS) task.c
                $(CC) $(CFLAGS) task.c

timer$(OSUF)    :       $(HDRS) timer.c
                $(CC) $(CFLAGS) timer.c

swap$(OSUF)     :
                if vax; \
                then $(CC) $(CFLAGS) vax_swap.s ; mv vax_swap.o swap$(OSUF) ; \
                else >swap$(OSUF) ; \
                fi

clean   :
                rm -f $(OBJS) *.i *..c

clobber :       clean
```

```
/* %Z% %M% %I% %H% %T% */
#include "../../incl/task.h"

object.~object()
{
        if (o_link) task_error(E_OLINK,this);
        if (o_next) task_error(E_ONEXT,this);
} /* delete */

/*      note that a task can be on a chain in several places
*/

/* object.remember() ... */

void object.forget(register task* p)
/* remove all occurrences of task* p from this object's task list */
{
        register olink* ll;
        register olink* l;

        if (o_link == 0) return;

        while (o_link->l_task == p) {
                ll = o_link;
                o_link = ll->l_next;
                delete ll;
                if (o_link == 0) return;
        };

        l = o_link;
        while (ll = l->l_next) {
                if (ll->l_task == p) {
                        l->l_next = ll->l_next;
                        delete ll;
                }
                else l = ll;
        };
}


void object.alert()
/* prepare IDLE tasks on this object for sceduling */
{
        register olink* l;

        for (l=o_link; l; l=l->l_next) {
                register task* p = l->l_task;
                if (p->s_state == IDLE) p->insert(0,this);
        }
}

void object.print(int n)
{
        int m = n & ~CHAIN;

        switch (o_type) {
```

```
        case QHEAD:
                ((qhead*) this)->print(m);
                break;
        case QTAIL:
                ((qtail*) this)->print(m);
                break;
        case TASK:
                ((task*) this)->print(m);
                break;
        case TIMER:
                ((task*) this)->print(m);
                break;
        default:
                printf("object (o_type==%d): ",o_type);
        }

        if (n&VERBOSE) {
                olink* l;
                printf("remember_chain:\n");
                for (l=o_link; l; l=l->l_next) l->l_task->print(m);
        }

        if (n&CHAIN) {
                if (o_next) o_next->print(n);
        }
        printf("\n");
}
```

```
/* %Z% %M% %I% %H% %T% */
#include "../../incl/task.h"

/*      a qhead's qh_queue has its pointer q_ptr pointing the last
        element of a circular list, so that q_ptr->o_next is the
        first element of that list.

   STRUCTURE:
        qhead <--> oqueue <--> qtail    (qhead and qtail are independent)
        oqueue --> circular queue of objects

        "pen and paper is recommended when trying to understand
        the list manipulations."
*/

/* construct qhead <--> (possible)oqueue --> 0 */
qhead.qhead(int mode, int max) : (QHEAD)
{
        if (0 < max) {
                qh_queue = new oqueue(max);
                qh_queue->q_head = this;
        };
        qh_mode = mode;
}


/* destroy q if not pointed to by a qtail */
qhead.~qhead()
{
        oqueue* q = qh_queue;

        if (q->q_tail)
                q->q_head = 0;
        else
                delete q;

}


/* remove and return object from head of q */
object* qhead.get()
{
        register oqueue* q = qh_queue;
11:
        if (q->q_count) {
                register object* oo = q->q_ptr;
                register object* p = oo->o_next;
                oo->o_next = p->o_next;
                p->o_next = 0;
                if (q->q_count-- == q->q_max) {
                        qtail* t = q->q_tail;
                        if (t) t->alert();
                };
                return p;
        }
```

```
        switch (qh_mode) {
        case WMODE:
                remember(thistask);
                thistask->sleep();
                forget(thistask);
                goto 11;
        case EMODE:
                task_error(E_GETEMPTY,this);
                goto 11;
        case ZMODE:
                return 0;
        }
}

/* create a tail for this queue */
qtail* qhead.tail()
{
        oqueue* q = qh_queue;
        register qtail* t = q->q_tail;

        if (t == 0) {
                t = new qtail(qh_mode,0);
                q->q_tail = t;
                t->qt_queue = q;
        }

        return t;
}


/* make room for a filter upstream from this qhead */
/* result:  (this)qhead<-->newq    (new)qhead<-->oldq ?<-->qtail? */
qhead* qhead.cut()
{
        oqueue* oldq = qh_queue;
        qhead* h = new qhead(qh_mode,oldq->q_max);
        oqueue* newq = h->qh_queue;

        oldq->q_head = h;
        h->qh_queue = oldq;

        qh_queue = newq;
        newq->q_head = this;

        return h;
}


/* this qhead is supposed to be upstream to the qtail t
   add the contents of this's queue to t's queue
   destroy this, t, and this's queue
   alert the spliced qhead and qtail if a significant state change happened
*/
void qhead.splice(qtail* t)
{
        oqueue* qt = t->qt_queue;
```

```
        oqueue* qh = qh_queue;

        int qtcount = qt->q_count;
        int qhcount = qh->q_count;
        int halert = (qtcount==0 && qhcount);   /* becomes non-empty */
        int talert = (qh->q_max <= qhcount && qhcount+qtcount<qt->q_max);
                                                /* becomes non-full */
        if (qhcount) {
                object* ooh = qh->q_ptr;
                object* oot = qt->q_ptr;
                qt->q_ptr = ooh;
                if (qtcount) {                  /* add the contents of qh to qt */
                        object* tf = oot->o_next;
                        oot->o_next = ooh->o_next;
                        ooh->o_next = tf;
                }
                qt->q_count = qhcount + qtcount;
                qh->q_count = 0;
        }

        (qh->q_tail)->qt_queue = qt;
        qt->q_tail = qh->q_tail;
        qh->q_tail = 0;

        delete t;
        delete this;

        if (halert) qt->q_head->alert();
        if (talert) qt->q_tail->alert();
}


/* insert new object at head of queue (after queue->q_ptr) */
int qhead.putback(object* p)
{
        oqueue* q = qh_queue;

        if (p->o_next) task_error(E_BACKOBJ,this);
l1:
        if (q->q_count++ < q->q_max) {
                if (q->q_count == 1) {
                        q->q_ptr = p;
                        p->o_next = p;
                }
                else {
                        object* oo = q->q_ptr;
                        p->o_next = oo->o_next;
                        oo->o_next = p;
                }
                return 1;
        }

        switch (qh_mode) {
        case WMODE:
        case EMODE:
                task_error(E_BACKFULL,this);
```

```
                goto 11;
        case ZMODE:
                return 0;
        }
}


void qhead.print(int n)
{
        oqueue* q = qh_queue;

        printf("qhead (%d): mode=%d, max=%d, count=%d, tail=%d\n",
                this,qh_mode,q->q_max,q->q_count,q->q_tail);

        if (n&VERBOSE) {
                int m = n & ~(CHAIN|VERBOSE);
                if (q->q_tail) {
                        printf("\ttail of queue:\n");
                        q->q_tail->print(m);
                }
                q->print(m);
        }
}


void oqueue.print(int n)
{
        object* p = q_ptr;

        if (q_count == 0) return;

        printf("\tobjectects on queue:\n");

        do {
                p->print(n);
                p = p->o_next;
        } while (p != q_ptr);

        printf("\n");
}
```

```
/* %Z% %M% %I% %H% %T% */
#include "../../incl/task.h"

/* construct qtail <--> oqueue */
qtail.qtail(int mode, int max) : (QTAIL)
{
        if (0 < max) {
                qt_queue = new class oqueue(max);
                qt_queue->q_tail = this;
        };
        qt_mode = mode;
}

/* destroy q if not also pointed to by a qhead */
qtail.~qtail()
{
        oqueue* q = qt_queue;

        if (q->q_head)
                q->q_tail = 0;
        else
                delete q;
}


/* insert object at rear of q (becoming new value of oqueue->q_ptr) */
int qtail.put(object* p)
{
        register oqueue* q = qt_queue;
l1:
        if (p->o_next) task_error(E_PUTOBJ,this);

        if (q->q_count < q->q_max) {
                if (q->q_count++) {
                        register object* oo = q->q_ptr;
                        p->o_next = oo->o_next;
                        q->q_ptr = oo->o_next = p;
                }
                else {
                        qhead* h = q->q_head;
                        q->q_ptr = p->o_next = p;
                        if (h) h->alert();
                }
                return 1;
        }

        switch (qt_mode) {
        case WMODE:
                remember(thistask);
                thistask->sleep();
                forget(thistask);
                goto l1;
        case EMODE:
                task_error(E_PUTFULL,this);
                goto l1;
        case ZMODE:
```

```
                return 0;
        }
}


/* create head for this q */
qhead* qtail.head()
{
        oqueue* q = qt_queue;
        register qhead* h = q->q_head;

        if (h == 0) {
                h = new qhead(qt_mode,0);
                q->q_head = h;
                h->qh_queue = q;
        };

        return h;
}


/* result:  ?qhead<-->? oldq<-->(new)qtail  newq<-->(this)qtail */
qtail* qtail.cut()
{
        oqueue* oldq = qt_queue;
        qtail* t = new qtail(qt_mode,oldq->q_max);
        oqueue* newq = t->qt_queue;

        t->qt_queue = oldq;
        oldq->q_tail = t;

        newq->q_tail = this;
        qt_queue = newq;

        return t;
}


/* this qtail is supposed to be downstream from the qhead h */
void qtail.splice(qhead* h)
{
        h->splice(this);
}


void qtail.print(int n)
{
        int m = qt_queue->q_max;
        int c = qt_queue->q_count;
        class qhead * h = qt_queue->q_head;

        printf("qtail (%d): mode=%d, max=%d, space=%d, head=%d\n",
                this,qt_mode,m,m-c,h);

        if (n&VERBOSE) {
                int m = n & ~(CHAIN|VERBOSE);
```

```
                if (h) {
                        printf("head of queue:\n");
                        h->print(m);
                }

                qt_queue->print(m);
        }
}
```

```
/* %Z% %M% %I% %H% %T% */
#include "../../incl/task.h"

void setclock(long t)
{
        if (clock) task_error(E_SETCLOCK,0);
        clock = t;
}

int in_error = 0;

int task_error(int n, object* oo)
{
        if (in_error)
                exit(in_error);
        else
                in_error = n;

        if (error_fct) {
                n = (*error_fct)(n,oo);
                if (n) exit(n);
        }
        else {
                print_error(n);
                exit(n);
        }
        in_error = 0;
        return 0;
}

char* error_name[] = {
        "",
        "object.delete(): has chain",
        "object.delete(): on chain",
        "qhead.get(): empty",
        "qhead.putback(): object on other queue",
        "qhead.putback(): full",
        "qtail.put(): object on other queue",
        "qtail.put(): full",
        "set_clock(): clock!=0",
        "schedule(): clock_task not idle",
        "schedule: terminated",
        "schedule: running",
        "schedule: clock<0",
        "schedule: task or timer on other queue",
        "histogram.new(): bad arguments",
        "task.save(): stack overflow",
        "new: free store exhausted",
        "task.new(): bad mode",
        "task.delete(): not terminated",
        "task.preempt(): not running",
        "timer.delete(): not terminated",
        "schedule: bad time",
        "schedule: bad object",
        "queue.delete(): not empty",
        "thistask->result()",
```

```
        "task.wait(thistask)",
};

void print_error(int n)
{
        register i = (n<1 || MAXERR<n) ? 0 : n;

        printf("\n\n***** task_error(%d) %s\n",n,error_name[i]);

        if (thistask) {
                printf("thistask: ");
                thistask->print(VERBOSE|STACK);
        }
        if (run_chain) {
                printf("run_chain:\n");
                run_chain->print(CHAIN);
        }
} /* task_error */

sched* run_chain = 0;
task* task_chain = 0;
long clock = 0;
task* thistask = 0;
task* clock_task = 0;
PFIO error_fct = 0;
/*PFIO sched_fct = 0;*/
PFV exit_fct = 0;

void sched.cancel(int res)
{
        if (s_state==RUNNING) remove();
        s_state = TERMINATED;
        s_time = res;
        alert();
}

int sched.result()
/* wait for termination and retrieve result */
{
        if (this == (sched*)thistask) task_error(E_RESULT,0);
        while (s_state != TERMINATED) {
                remember(thistask);
                thistask->sleep();
                forget(thistask);
        }

        return (int) s_time;
}

void sched.schedule()
/* schedule either clock_task or front of run_chain */
{
        register sched* p;
        register long tt;
```

```
111:
        if (p = run_chain) {
                run_chain = (sched*) p->o_next;
                p->o_next = 0;
        }
        else {
                if (exit_fct) (*exit_fct)();
                exit(0);
        }

        tt = p->s_time;
        if (tt != clock) {
                if (tt < clock) task_error(E_SCHTIME,this);
                clock = tt;
                if (clock_task) {
                        if (clock_task->s_state != IDLE)
                                task_error(E_CLOCKIDLE,this);
                        /* clock_task preferred */
                        p->o_next = (object*) run_chain;
                        run_chain = p;
                        p = (sched*) clock_task;
                }
        }

        switch (p->o_type) {
        case TIMER: /* time is up; "delete" timer & schedule next task */
                p->s_state = TERMINATED;
                p->alert();
                goto 111;
        case TASK:
                if (p != this) {
                        if (thistask && thistask->s_state != TERMINATED)
                                thistask->save();
                        thistask = (task*) p;
                        thistask->restore();
                }
                break;
        default:
                task_error(E_SCHOBJ,this);
        }
} /* schedule */

void sched.insert(int d, object* who)
/*
        schedule THIS to run in ``d'' time units
        inserted by who
*/
{
        register sched * p;
        register sched * pp;
        register long tt = s_time = clock + d;

        switch (s_state) {
        case TERMINATED:
                task_error(E_RESTERM,this);
                break;
```

```
        case IDLE:
                break;
        case RUNNING:
                if (this != (class sched *)thistask) task_error(E_RESRUN,this);
        }

        if (d<0) task_error(E_NEGTIME,this);

        if (o_next) task_error(E_RESOBJ,this);

        s_state = RUNNING;
        if (o_type == TASK) ((task *) this)->t_alert = who;

        /* run_chain ordered by s_time */
        if (p = run_chain) {
                if (tt < p->s_time) {
                        o_next = (object*) run_chain;
                        run_chain = this;
                }
                else {
                        while (pp = (sched *) p->o_next) {
                                if (tt < pp->s_time) {
                                        o_next = pp;
                                        p->o_next = this;
                                        return;
                                }
                                else p = pp;
                        }
                        p->o_next = this;
                }
        }
        else
                run_chain = this;
}

void sched.remove()
/* remove from run_chain and make IDLE */
{
        register class sched * p;
        register class sched * pp;

        if (p = run_chain)
                if (p == this)
                        run_chain = (sched*) o_next;
                else
                        for (; pp = (sched*) p->o_next; p=pp)
                                if (pp == this) {
                                        p->o_next = pp->o_next;
                                        goto 11;
                                }
11:
        s_state = IDLE;
        o_next = 0;
}

void sched.print(int n)
```

```
{
        int m = n & ~CHAIN;

        switch (o_type) {
        case TIMER:
                ((timer*)this)->print(m);
                break;
        case TASK:
                ((task*)this)->print(m);
                break;
        }

        if (n&CHAIN) {
                if (o_next) ((sched*) o_next)->print(n);
        }
}
```

```
/* %Z% %M% %I% %H% %T% */
#include "../../incl/task.h"

histogram.histogram(int nb, int ll, int rr)
{
        register int i;
        if (rr<=ll || nb<1) task_error(E_HISTO,0);
        if (nb%2) nb++;
        while ((rr-ll)%nb) rr++;
        binsize = (rr-ll)/nb;
        h = new int[nb];
        while (h == 0) task_error(E_STORE,0);
        for (i=0; i<nb; i++) h[i] = 0;
        l = ll;
        r = rr;
        nbin = nb;
        sum = 0;
        sqsum = 0;
}

void histogram.add(int a)
/* add a to one of the bins, adjusting histogram, if necessary */
{
        register int i, j;

        /* make l <= a < r, */
        /*      possibly expanding histogram by doubling binsize and range */
        while (a<l) {
                l -= r - l;
                for (i=nbin-1, j=nbin-2; 0<=j; i--, j-=2) h[i] = h[j] + h[j+1];
                while(i >= 0) h[i--] = 0;
                binsize += binsize;
        }
        while (r<=a) {
                r += r - l;
                for (i=0, j=0; i<nbin/2 ; i++, j+=2) h[i] = h[j] + h[j+1];
                while (i < nbin) h[i++] = 0;
                binsize += binsize;
        }
        sum += a;
        sqsum += a * a;
        h[(a-l)/binsize]++;
}

void histogram.print()
/*
        printout non-empty ranges
*/
{
        register int i;
        register int x;
        int d = binsize;

        for (i=0; i<nbin; i++) {
                if (x=h[i]) {
                        int ll = l+d*i;
```

```
                        printf("[%d:%d) : %d\n",ll,ll+d,x);
                }
        }
}

/*
int erand.draw()
{
        int k;
        float a;

        for(k=0;;k++) {
                register float u1, u2;
                a = u1 = fdraw();
                do {
                        u2 = fdraw();
                        if (u1 < u2) return (int) k+a;
                        u1 = fdraw();
                } while (u1<u2);
        }

}
*/
```

```
/* %Z% %M% %I% %H% %T% */

#include "../../incl/task.h"

/*      macros giving the addresses of the stack frame pointer
        and the program counter of the caller of the current function
        given the first local variable

        TOP points to the top of the current stack frame
        given the last local variable
*/

#ifdef pdp11

#define FP()            (&_that+4)
#define OLD_FP(fp)      (*fp)
#define TOP(var9)       (&var9)

#else
/* This used to be  an if vax.  There should probably be a change
for the 3B's */

#define FP(p)           ((int*)(&p+1))
#define OLD_AP(fp)      (*(fp+2))
#define OLD_FP(fp)      (*(fp+3))
#define TOP(p)          top(&p)
extern int * top(...);

#endif

#define SETTRAP()       t_trap = *(t_basep-t_stacksize+1)
#define CHECKTRAP()     if (t_trap != *(t_basep-t_stacksize+1)) task_error(E_STACK,


int _hwm;

class team
{
friend task;
        int     no_of_tasks;
        task*   got_stack;
        int*    stack;
        team(task*, int = 0);
        ~team() { delete stack; }
};
team.team(task* t, int stacksize) {
        no_of_tasks = 1;
        got_stack = t;
        if (stacksize) {
                stack = new int[stacksize];
                while (stack == 0) task_error(E_STORE,0);
        }
}


void usemainstack()
```

```
/* fudge to allow simple stack overflow check */
{
        register v[SIZE+100];

        if (_hwm)
                for (register i=0;i<SIZE+100;i++) v[i] = UNTOUCHED;
        else
                v[0] = 0;
}

void copy_stack(register* f, register c, register* t)
/*
        copy c words down from f to t
        do NOT attempt to copy "copy_stack"'s own stackframe
*/
{
        while (c--) { *t-- = *f--;}
}

void task.swap_stack(int* p, int* ta, int* de, int* pa, int* ap)
{
        int x = pa-TOP(x)+1;    /* size of active stack */
        copy_stack(pa,x,p);
        x = pa-p;               /* distance from old stack to new */
        t_framep = ta-x;        /* fp on new frame */
                                /* now doctor the new frame */
#ifdef vax
        OLD_AP(t_framep) = int(ap-x);
#endif
        OLD_FP(t_framep) = int(de-x);
        restore();
}

task.task(char* name, int mode, int stacksize) : (TASK)
/*
        executed in the task creating a new task - thistask.
        1:      put thistask at head of scheduler queue,
        2:      create new task
        3:      transfer execution to new task
        derived::derived can never return - its return link is destroyed

        if thistask==0 then we are executing on main()'s stack and
        should turn it into the "main" task
*/
{
        int* p;
        int* ta_fp = (int*)FP(p);
        int* de_fp = (int*)OLD_FP(ta_fp);
#ifndef pdp11
/* xxx changed from ifdef vax */
        int* de_ap = (int*)OLD_AP(ta_fp);
#endif
        int* pa_fp = (int*)OLD_FP(de_fp);
        int x;

        t_name = name;
```

```
        t_mode = (mode) ? mode : DEDICATED;
        t_stacksize = (stacksize) ? stacksize : SIZE;
        t_size = 0;             /* avoid stack copy at initial restore */
        t_alert = 0;
        s_state = RUNNING;
        t_next = task_chain;
        task_chain = this;
        th = this;      /* fudged return value -- "returned" from swap */

        switch ((int)thistask) {
        case 0:
                /* initialize task system by creating "main" task */
                thistask = (task*) 1;
                thistask = new task("main");
                break;
        case 1:
                /*      create "main" task      */
                usemainstack();         /* ensure that store is allocated */
                t_basep = (int*)OLD_FP(pa_fp);  /* fudge, what if main
                                                   is already deeply nested
                                                */
                t_team = new team(this);        /* don't allocate stack */
                t_team->no_of_tasks = 2;        /* never deallocate */
                return;
        }
        thistask->th = this;    /* return pointer to "child" */
        thistask->t_framep = de_fp;
        thistask->insert(0,this);

        switch (t_mode) {
        case DEDICATED:
                t_team = new team(this,t_stacksize);
                t_basep = t_team->stack + t_stacksize - 1;
                if (_hwm) for (x=0; x<t_stacksize; x++)
                        t_team->stack[x] = UNTOUCHED;
                thistask = this;
                swap_stack(t_basep,ta_fp,de_fp,pa_fp,de_ap);
        case SHARED:
                thistask->t_mode = SHARED; /* you cannot share on your own */
                t_basep = pa_fp;
                t_team = thistask->t_team;
                t_team->no_of_tasks++;
                t_framep = ta_fp;
                if (mode==0 && stacksize==0)
                        t_stacksize = thistask->t_stacksize - (thistask->t_basep -
                thistask = this;
                return;
        default:
                task_error(E_TASKMODE,this);
        }
}

void task.save()
/*
        save task's state so that ``restore'' can resume it later
        by returning from the function which called "save"
```

```
                - typically the scheduler
*/
{
        int* x;
        register* p = (int*)FP(x);

        t_framep = (int*)OLD_FP(p);

        CHECKTRAP();

        if (t_mode == SHARED) {
                register int sz;
                t_size = sz = t_basep - p + 1;
                p = new int[sz];
                while (p == 0) task_error(E_STORE,0);
                t_savearea = &p[sz-1];
                copy_stack(t_basep,sz,t_savearea);
        };
}

extern int rr2,rr3,rr4;
int rr2,rr3,rr4;

swap(task*);
sswap(task*);

void task.restore()
/*
        make "this" task run after suspension by returning from the frame
        denoted by "t_framep"

        the key function "swap" is written in assembly code,
        it returns from the function which "save"d the task
                - typically the scheduler

        "sswap" copies the stack back from the save area before "swap"ing
        arguments to "sswap" are passed in rr2,rr3,rr4 to avoid overwriting them
        it is equivallent to "copystack" followed by "swap".
*/
{
        register sz;

        SETTRAP();

        if ((t_mode == SHARED) && (sz=t_size)){
                register* p = t_savearea - sz + 1;
                register x = (this != t_team->got_stack);
                t_team->got_stack = this;
                delete p;
                if (x) {
                        rr4 = (int) t_savearea;
                        rr3 = sz;
                        rr2 = (int) t_basep;
                        sswap(this);
                }
                else
```

```
                        swap(this);
        }
        else
                swap(this);
}

void task.cancel(int val)
/*
        TERMINATE and free stack space
*/
{
        sched::cancel(val);
        if (_hwm) t_size = curr_hwm();
        if (t_team->no_of_tasks-- == 1) delete t_team;
}

task.~task()
/*
        free stack space and remove task from task chain
*/
{
        if (s_state != TERMINATED) task_error(E_TASKDEL,this);
        if (this == task_chain)
                task_chain = t_next;
        else {
                register task* t;
                register task* tt;

                for (t=task_chain; tt=t->t_next; t=tt)
                        if (tt == this) {
                                t->t_next = t_next;
                                break;
                        }
        }

        if (this == thistask) {
                delete (int*) thistask; /* fudge: free(_that) */
                thistask = 0;
                schedule();
        }
}

void task.resultis(int val)
{
        cancel(val);
        if (this == thistask) schedule();
}

void task.sleep()
{
        if (s_state == RUNNING) remove();
        if (this == thistask) schedule();
}

void task.delay(int d)
{
```

```
        insert(d,this);
        if (thistask == this) schedule();
}

int task.preempt()
{
        if (s_state == RUNNING) {
                remove();
                return s_time-clock;
        }
        else {
                task_error(E_TASKPRE,this);
                return 0;
        }
}

char* state_string(int s)
{
        switch (s) {
        case IDLE:              return "IDLE";
        case TERMINATED:        return "TERMINATED";
        case RUNNING:           return "RUNNING";
        default:                return 0;
        }
}

char* mode_string(int m)
{
        switch(m) {
        case SHARED:            return "SHARED";
        case DEDICATED:         return "DEDICATED";
        default:                return 0;
        }
}

void task.print(int n)
/*
        ``n'' values:   CHAIN,VERBOSE,STACK
*/
{
        char* ss = state_string(s_state);
        char* ns = (t_name) ? t_name : "";

        printf("task %s ",ns);
        if (this == thistask)
                printf("(is thistask):\n");
        else if (ss)
                printf("(%s):\n",ss);
        else
                printf("(state==%d CORRUPTED):\n",s_state);

        if (n&VERBOSE) {
                int res = (s_state==TERMINATED) ? (int) s_time : 0;
                char* ms = mode_string(t_mode);
                if (ms == 0) ms = "CORRUPTED";
                printf("\tthis==%d mode=%s alert=%d next=%d result=%d\n",
```

```
                        this,ms,t_alert,t_next,res);
        }

        if (n&STACK) {
                printf("\tstack: ");
                if (s_state == TERMINATED) {
                        if (_hwm) printf("hwm=%d",t_size);
                        printf(" deleted\n");
                }
                else {
                        int b = (int) t_basep;
                        int x = ((this==thistask) || t_mode==DEDICATED) ? b-(int)t_
                        printf("max=%d current=%d",t_stacksize,x);
                        if (_hwm) printf(" hwm=%d",curr_hwm());
                        printf(" t_base=%d, t_frame=%d, t_size=%d\n",b,t_framep,t_s
                }
        }

        if (n&CHAIN) {
                if (t_next) t_next->print(n);
        }
}

int task.curr_hwm()
{
        int* b = t_basep;
        int i;
        for (i=t_stacksize-1; 0<=i && *(b-i)==UNTOUCHED; i--) ;
        return i;
}

int task.waitlist(object* a)
{
        return waitvec(&a);
}

int task.waitvec(object* * v)
/*
        first determine if it is necessary to sleep(),
        return hint: who caused return
*/
{
        int i = 0;
        int r;
        object* ob;

        while (ob = v[i++]) {
                t_alert = ob;
                switch (ob->o_type) {
                case TASK:
                case TIMER:
                        if (((sched*)ob)->s_state == TERMINATED) goto ex;
                        break;
                case QHEAD:
                        if (((qhead*)ob)->rdcount()) goto ex;
                        break;
```

```
                case QTAIL:
                        if (((qtail*)ob)->rdspace()) goto ex;
                        break;
                }
                ob->remember(this);
        }
        if (i==2 && v[0]==(object*)thistask) task_error(E_WAIT,0);
        sleep();
ex:
        i = 0;
        while (ob = v[i++]) {
                ob->forget(this);
                if (ob == t_alert) r = i-1;
        }
        return r;
}
```

```
/* %Z% %M% %I% %H% %T% */
#include "../../incl/task.h"

timer.timer(int d) : (TIMER)
{
        s_state = IDLE;
        insert(d,this);
}

timer.~timer()
{
        if (s_state != TERMINATED) task_error(E_TIMERDEL,this);
}

void timer.reset(int d)
{
        remove();
        insert(d,this);
}

void timer.print(int n)
{ n;/*avoid warning*/
        long tt = s_time;
        printf("timer %ld == clock+%ld\n",tt,tt-clock);
}
```

```
# %Z% %M% %I% %H% %T%
        #       swap of SHARED

.globl _rr4
.globl _rr3
.globl _rr2

        .globl _sswap
        .align  1
_sswap:
        .word   0x0000
        movl    4(ap),r1        # this
        movl    _rr4,r4
        movl    _rr3,r3
        movl    _rr2,r2

L1:
        tstl    r3
        jeql    L2
        decl    r3
        movl    (r4),(r2)
        cmpl    -(r4),-(r2)
        jbr     L1
L2:
        #  the following constant is the displacement of t_framep in task
        movl    20(r1),fp       #       fp = this->t_framep
        movl    24(r1),r0       #       fudge return -- this->th
        ret

.globl _swap
        .align  1
_swap:
        .word   0x0000
        movl    4(ap),r1        #       r1 = this
        #  the following constant is the displacement of t_framep in task
        movl    20(r1),fp       #       fp = this->t_framep
        movl    24(r1),r0       #       fudge return -- this->th
        ret


        .globl  _top
        .align  1
_top:
        .word   0x0000
        addl3   $1,(ap),r0
        ashl    $2,r0,r0
        addl2   ap,r0
        ret
```