Model for Tuberculosis in the UK

This individual-based model (IBM) is to simulate tuberculosis dynamics in the UK. This version of the model stores individual strain types for each infection to simulate strain type clustering patterns seen in disease cases. This version was developed for use in the West Midlands, a region with a population size of around five million people. One major characteristic of individuals is their region of birth, UK or non-UK. In the `SSAV` version of the model, the non-UK-born region of birth is divided into Sub-Saharan African born (SSA-born) and other non-UK born (ONUK-born) for some model parameters.

1. BACKGROUND ALGORITHMS. The core IBM algorithms, external to this simulation program, were written by Clarence Lehman (CL) in 2009. The method was first developed for simulation of HIV dynamics in the US. Beginning in January 2010, and with initial help from CL, Adrienne Keen (AK) adapted this IBM skeleton for modelling tuberculosis dynamics in the UK, first for fitting the model to tuberculosis notifications in England and Wales and then for simulating genotyping data from the West Midlands.

2. SUMMARY OF METHOD. This is an event-based simulation where continuous time is simulated directly. There is no arbitrary time step. Instead, events are processed one at a time, chronologically. The time variable `t` is time in years, with arbitrarily high resolution down to small fractions of a instant and all complexities and inaccuracies associated with multiple events during a finite time step vanish – such as undershooting zero when the sum of the rates times the width of the time step exceeds unity. Continuous time also allows all activities to occur in a single data array, rather than having to swap old and new arrays at each time step. Events are assumed to following probability distributions that vary through time and space.

States of the system never change spontaneously—all changes are induced by some other event in the system and usually scheduled in advance. The scheduled times are determined stochastically from functions whose characteristics may depend on the state of the individual and the environment at the time. An individual's age, sex, infection history, or any other considerations can be incorporated into the functions.

For example, death is scheduled at the time of birth, with the time chosen randomly from a life-span distribution for babies born in the simulated year. But the scheduled time of death is not immutable, nor are any other scheduled times in the system. If the individual develops disease, the scheduled time of death may be cancelled and a new time of death due to tuberculosis may be scheduled instead. At no time does the program visit an individual when it does not need to, and therein lies its speed.

Many future events may apply to each individual and are saved for that individual, but only the earliest among each individual's events enters a global "list of future events."

3. MAIN DATA STRUCTURES. Each individual is assigned a number 1 through `n` and recorded in a linear array `A` of structures `Indiv`. Each element of `A` defines the state of the correspondng individual.

For example, this would be defined as `struct A[indiv+3]`, the main array of individuals. Suppose there are 3 UK-born and 3 non-UK-born individuals, with a total maximum population size of 14 (`indiv`). Note, in the `SSAV` version of the model, SSAs are stored as non-UK born and it is not possible to tell from their ID number alone whether they are SSA-born or ONUK-born.

```
 n   A[n]                          A array index
-------------------------------------------------
 0  [Reserved]              (Null pointer for list)
 1  (Non-UK-born)
 2  (Non-UK-born)
 3  (Non-UK-born)                   immid-1
 4  [Empty]                         immid
 5  [Empty]
 6  [Empty]
 7  [Empty]                         maximm
-------------------(Imaginary separator, non-UK/UK born)
 8  (UK-born)                       maximm+1
 9  (UK-born)
10  (UK-born)                       ukbid-1
11  [Empty]                         ukbid
12  [Empty]
13  [Empty]
14  [Empty]                         indiv
15  External event, birth           indiv+1 or BIRTH
16  External event, immigration     indiv+2 or IMM
-------------------------------------------------
```

4. MODEL FITTING. Prior versions of the model were designed to work with an optimization algorithm for model fitting, currently found in `fit5.c`. In this version of the model, the fitting routine is not needed.

To run this program stand-alone, as opposed to inside the fitting routine, simply comment out the `define main mainiac` and everything else will be handled automatically.

Below is a sample program call when it is running as an individual executable, not linked with the fitting routine. A different syntax is used for the call inside `fit5i.c`.

```
tb36gen  df=2.5 d1uk20=0.10 d2uk20=0.0003 d3uk20=0.05
```

When linked with the fitting routine, the model is called from the fitting routine, not as an independent executable, so that it is compatible with parallel runs using MPI commands. In this set up, the model accepts four variable disease risk parameters, `df`, `d1uk20[M]`, `d2uk20[M]`, and `d3uk20[M]`. See the function `Data` for information on these. Briefly, `df` is the factor by which UK-born disease risks are multiplied to obtain non-UK born disease risks. The other three parameters are UK-born disease risks for Primary, Reactivation, and Reinfection Disease respectively, in adult males (those aged 20 years and over). In this version of the model, disease risk are fixed for children under ten years of age, allowing for fewer variable parameters.

5. OTHER. Note that the following program substitutes the term `dec` for the C term `double`. It is short for "decimal", in contrast and parallel with "integer", saving valuable coding columns at the left of the line and helping data names line up.

The sequence of random numbers is specified on the command line with phrases like `randseq=0`, `randseq=1`, `randseq=-6`, `randseq=239702397623`, and so forth. Fixed sequences that are the same each time the program runs occur when `randseq` is 0 or greater. Each integer gives a different sequence of random numbers. (Actually, it gives only a different

starting point in a single long sequence of random numbers.) Positive integers, or zero, are typically used in testing because program results are precisely repeatable.

Arbitrary sequences that are different, with high probability, each time the program runs occur when `randseq` is negative. The date and time, measured to the nearest second, selects the starting sequence, then the negative value modifies that sequence. Thus if several instances of the program were started on separate processors at the same time, the first with `randseq=-1`, the second with `randseq=-2`, and so forth, each instance of the program is guaranteed a different random number sequence. Unlike the case with non-negative integers, however, the sequence be different each time the program runs, with very high probability.

The actual starting seed, incorporating the time of day if requested by a negative value of `randseq`, is stored in `rand0` and reported at the end of the run. That allows a run to be repeated exactly even if it was started with an arbitrary sequence.

6. NOTES ON GENETIC STRAINS. At model initialization, strains are randomly assigned from two different strain type distributions derived from empirical data, one for non-UK born and one for UK-born, saved in `sdimm` and `sduk`, respectively. Also, the total number of available strains in each distribution is `is0` and `is1`, respectively. These are set in define statements before the model is run to faciliate array initialization and contiguous number of strains. New mutant strains which appear as the simulation runs will be given strain IDs distinct from any in the `sdimm` or `sduk` distributions. These IDs will begin with integer `is0+is1` and will always be greater than or equal to this value. The variable `stid` holds the next available strain type ID for new mutants. The diagram below illustrates which strain IDs belong to which individuals in the simulation:

StrainID	Available for	Generic Reference
0	(Not used)	
1	Initial non-UK, migrants	
2	Initial non-UK, migrants	
3	Initial non-UK, migrants	
4	Initial non-UK, migrants	is0
5	Initial UK	is0+1
6	Initial UK	
7	Initial UK	
8	Initial UK	is0+is1
9	New mutant strain	is0+is1+1
10	New mutant strain	
11	New mutant strain	
14	Next available new mutant	stid

```
include <stdio.h>
include <stdlib.h>
include <time.h>
include "common.h"
include "fileio.h"
```

```
define PN (q1+1)              Number of elements in array N.
define T0 1981                Start time of model, years.
define T1 2012                End time of model, years. The simulation
                                ends *before* reaching this year.
define TDATA 2007             First year of observed data, for reporting times.
define SSAV   1               Switch model version depending on existence
                                of separate SubSaharan African group,
                                0=non-SSA, 1=SSA.
define SUPER  1               Flag for whether model is run on supercomputer,
```

```
                                  0=no, 1=yes (changes population sizes).
define DPARAM 1                   Allows model to accept disease progression
                                    parameters (4 in this version), 0=no, 1=yes.
define REC    1                   Flag for whether tallying recent transmissions.
define BIRTH (indiv+1)            Index used for scheduling births.
define IMM   (indiv+2)            Index for scheduling arrival of immigrants.
define RT    (T1-T0)              Running time of model, calendar years.
define NUK   0                    Array index for non-UK born.
define UK    1                    Array index for UK-born.
define HIV   2                    Array index for HIV+.
define SSA   2                    Array index for SSA-born.
define M     0                    Array index for males.
define F     1                    Array index for females.
define E     0.0000000001;        Small number added to some event times to
                                    ensure they happen in the future.
define AC 122                     Age classes for mortality data.
define LAT 5                      Years to Remote from recent (re)infection.
define BY (2012-1870+1)           Number of birth cohorts for mortality data.
define IS0 5000                   Number of strains for migrants.
define IS1 1000                   Number of strains for UK-born at model
                                    initialization.
dec N[PN];                        Current number in each disease state.
dec N2[4][2][2][RT];              Population sizes in the model at end of year by
                                    age, sex, rob and year.
dec N3[4][2][2][RT];              Population sizes observed, which are compared
                                    with model population sizes and used to correct
                                    case numbers produced by the model.
int Np[4][2][2][RT];              Infected persons by age, sex, rob, year.
dec age1[2],age2[2],agec[2];      Accumulators for 1st and 2nd moments of age.
dec repc[4][2][2][2][RT];         Reported cases by age category, sex, rob,
                                    disease site and year.
dec repc2[4][2][2][2][RT][5];     Time/place of tranmission for reported cases,
                                    indexed as repc plus last index is 0 for
                                    total cases, 1 for recent/UK, 2 for older/UK
                                    3 for recent/NonUK and 4 for older/NonUK.
dec repc3[15000][7];              Data on typed cases, 0=age, 1=sex, 2=rob, 3=time
                                    of report, 4=place/time of infection, 5=strain ID,
                                    6=number of others with identical strain.
int ari[2][RT];                   Number of successful transmissions, by rob
                                    (UK/NUK) and year.
int clust[4][2][2][5];            Clustering data by age, sex, and rob, where
                                    last index gives cases which are: 0=unique and
                                    non-recent/non-UK,1=unique and recent/UK,
                                    2=clustered and non-recent/non-UK, 3=clustered
                                    and recent/UK, 4=total typed cases.
int deaths;                       Current number of deaths.
int events;                       Current number of events dispatched.
int immid;                        Next available ID number for immigrants.
int ukbid;                        Next available ID number for UK-born.
int stid;                         Next available ID for new strain types.
int repid;                        Next available ID for repc3 case report.
extern dec t;                     Current time (Managed by EventSchedule).
dec pt;                           Time of previous report.
dec t0 = T0;                      Beginning time of simulation.
dec t1 = T1;                      End time of simulation.
```

`int tdata = TDATA;`	First year of observed data.
`int lup;`	Year of last update to birth and immigration rates, which are sensitive to calendar year.
`dec runid;`	ID number for printing output files.
`unsigned long startsec;`	Starting clock time, seconds of Unix.
`unsigned long rand0;`	Starting random number seed.
`struct Indiv *A;`	State of each individual, including their characterisitics, saved event times, etc.

1.1 Parameters and control variables

Population initialization

`int maximm;`	Maximum immigrants in pop'n at any time.
`dec inf1981[121][3][2][9];`	Cumulative probabilities of the 9 disease states for pop. initialization (by a,s,rob).
`dec n1981[121][2][2];`	Numbers in each age/sex/rob category at population initialization, 1981.
`dec ssa1981[121][2];`	Proportion SSA by age/sex category.

Infection transmission

`dec c[2][2];`	Effective contacts per year per pulmonary case (smear+) by sex and region of birth.
`dec pcc;`	Probability effective contact is close contact (drawn from within own region of birth, UK or non-UK).
`dec  s2[2];`	Relative susceptibility to reinfection (s).
`dec smear[121];`	Proportion smear positive by age.

Vaccination

`dec  v1[2];`	Efficacy of vaccine (rob).
`dec  v2[2];`	Portion vaccinated at designated age (rob).
`dec  v3[2];`	Average age of vaccination (rob).

Disease progression

`dec d1[2][3][121];`	Proportion Recently Infected who progress to disease over first 5 years of infection (by sex,rob,age)
`dec d3[2][3][121];`	Proportion Reinfected who progress to disease over first 5 yrs of reinfection (by a,s,r).
`dec drr[6];`	Cumulative, relative risk of disease progression by year since infection, used with `d1` and `d3` (for first 5 years of infection).
`dec B1[6];`	Array of values for finding cumulative risk of in first five years of infection/reinfection, used with drr.
`dec d2[2][3][AC+2];`	Proportion of those Remotely Infected who progress to disease, cumulative dsn by sex,rob (where r=0, 1, or 2. 2=HIV+ and SSA in SSA version of model) and age.
`dec A2[AC+2];`	Array of values for finding random time to disease for Remote Infection.

`dec ehiv;`	Factor by which non-UK born disease risks are multiplied for HIV+ SSA individuals.
`dec df;`	Factor by which UK-born disease progression rates are multiplied to get immigrant rates.
`dec d1uk10[2];`	Rates of disease progression for primary (1),
`dec d1uk20[2];`	Reactivation (2), and Reinfection (3) disease
`dec d2uk10[2];`	for those aged 0-10 (10) and 20+ (20) by sex.
`dec d2uk20[2];`	These help construct `d1`, `d2`, and `d3`.
`dec d3uk10[2];`	
`dec d3uk20[2];`	
`dec sdf1[2];`	Risk ratios for female:male disease
`dec sdf2[2];`	progression risks/rates (by age 0-10,20+).
`dec sdf3[2];`	
`dec presp;`	Proportion of all tuberculosis which is respiratory, for correcting disease risks in Vynn. and Fine to pulmonary disease risks in children.
`dec p1[121][2][2];`	Portion pulm -primary disease (a,s,rob)
`dec p2[121][2][2];`	Portion pulm -reactivation disease (a,s,rob)
`dec p3[121][2][2];`	Portion pulm -reinfection disease (a,s,rob)
`dec duk1p[2][2];`	Intermediate parameters for pulmonary-only
`dec duk2p[2][2];`	rates of disease progression. Used for
`dec duk3p[2][2];`	incorporating estimated rates from Vynnycky and Fine into rates for this model, which are combined pulmonary/non-pulmonary. Indexed by age (0-10yrs, 20+yrs) and sex.
`dec d1p[121][2][2];`	Intermediate param's for pulmonary-only rates
`dec d2p[121][2][2];`	of disease progression. Used to get overall
`dec d3p[121][2][2];`	rates using those estimated by Vynn. and Fine (which are pulmonary rates). Indexed by age, sex, and rob. Precursors to `d1`, `d2`, `d3`.

Disease recovery, indexed by sex

`dec r3[2];`	Primary disease recovery rate
`dec r4[2];`	Reactivation disease recovery rate
`dec r5[2];`	Reinfection disease recovery rate
`dec r6[2];`	Primary non-pulmonary disease recovery rate
`dec r7[2];`	Reactivation non-pulm. disease recovery rate
`dec r8[2];`	Reinfection non-pulm. disease recovery rate

Mortality

`dec A1[AC];`	Holds ages 0-121 which correspond to the cumulative probabilities in M1.
`dec M1[BY][2][AC];`	Cumulative probabilities of death by a given birth cohort, sex and age.
`dec cft[121][2][RT];`	Case fatality rate due to tuberculosis (a,type dis,y)

Below are mortality rates used to generate lifetimes with exponential distribution, used for "reduction testing" of the model.

`dec m1 [2][RT];`	Mortality of uninf/vacc/inf ind's (sex, y)
`dec m6 [2][RT];`	Mortality of primary disease (sex,y)
`dec m7 [2][RT];`	Mortality of reactivation disease (sex,y)
`dec m8 [2][RT];`	Mortality of reinfection disease (sex,y)
`dec m9 [2][RT];`	Mortality of primary non-pulm. disease (sex,y)
`dec m10[2][RT];`	Mortality of reactivated non-pulm. disease (s,y)

`dec m11[2][RT];`	Mortality of reinfection non-pulm. disease (s,y)

Birth and migration

`dec bcy[RT];`	Births by calendar year.
`dec pmale[RT];`	Portion of newborns who are male by year.
`dec immig[RT];`	Total (uk+non-uk-born) immigrants by year.
`dec pimm[RT];`	Proportion of immigrants non-UK born by year.
`dec ssaim[RT];`	Proportion of non-UK born immigrants born in SubSaharan Africa by year.
`dec hivp[2][RT];`	HIV Prevalence in SubSaharan African born immigrants by sex,year.
`dec immsex[RT][3];`	Proportion immigrants who are male by yr and rob:0=non-UK,1=UK, 2=non-UK SSA. Note there are different input files for SSA and non-SSA version of the model.
`dec immage[RT][2][3][7];`	Cumulative proportion of immigrants (by yr, sex,rob) in age classes, for use with RandF.
`dec immageX[RT][2][3][6];`	Probabilities of 6 age classes (by yr,sex, rob) from ONS inflow data, —as in `immsex` note there are 2 versions of the input file for this array. Precursor to `immage`.
`dec infimm[121][3][RT][9];`	Cumulative probabilities immigrants enter disease states, by age,rob and year.
`dec Ax[9];`	State variables which accompany `infimm`.
`dec ypb, ypi;`	Years per birth, years per immigrant.
`dec em[2][3];`	Annual emigration rate by sex, rob.

Strain type related

`dec md;`	Mutations per year per strain type (diseased).
`dec mi;`	Mutations per year per strain (infected).
`dec is0 = IS0;`	Number of strains for migrants.
`dec is1 = IS1;`	Number of strains for UK-born.
`dec S0[IS0+1];`	Strain IDs to accompany strain distribution for migrants.
`dec S1[IS1+1];`	Strain IDs to accompany strain distribution UK-born at initialization.
`dec sdimm[IS0+1];`	Cumulative probabilities of strain types for non-UK born and migrants.
`dec sduk[IS1+1];`	Cumulative probabilities of strain types for UK-born at initialization.
`dec ptyped[2];`	Proportion of cases with strain type, by disease site, 0=non-pulm, 1=pulm.

Assorted

The following two recovery rates will not be used in version of model that defines Remote Infection as `LAT` years after most recent infection.

`dec r1[2];`	Rate Recent Infection moves to remote (s)
`dec r2[2];`	Rate Reinfection moves to remote (s)
`dec proprep;`	Proportion of cases reported.
`dec relativetime = 0;`	Set for relative time reporting.
`dec randseq = 0;`	Random number sequence (set with `randseq=N`).

```
dec tgap    = 0.5;              Time between reports, years.
dec kernel  = 0;                Contagion kernel, 0=Panmictic, 1=Cauchy.
dec sigma   = 1;                Width of contagion kernel, where applicable.

struct IO fmt[] =               Format statements for input/output.
{ /*00*/ { (dec*)bcy,     {-'i',RT} },
  /*01*/ { (dec*)immig,   {-'i',RT} },
  /*02*/ { (dec*)pimm,    {-'i',RT} },
  /*03*/ { (dec*)ssaim,   {-'i',RT} },
  /*04*/ { (dec*)pmale,   {-'i',RT} },
  /*05*/ { (dec*)hivp,    {-'s',2,-'Y',RT}, {-'y',-'S'} },
  /*06*/ { (dec*)infimm,  {-'a',121,-'r',3,-'y',RT,-'q',9}, {-'R',0,SSAV+1,-'Y',-'Q',-'A'} },
  /*--*/ { (dec*)inf1981, {-'a',121,-'r',2,-'q',9},  {-'a',120,0,-'r',UK,UK,  -'q',1,7} },
  /*--*/ { (dec*)inf1981, {-'a',121,-'r',2,-'q',9},  {-'a',120,0,-'r',NUK,NUK,-'q',1,7} },
  /*09*/ { (dec*)ssa1981, {-'a',121,-'s',2}, {-'s',-'A'} },
  /*10*/ { (dec*)n1981,   {-'a',121,-'s',2,-'r',2},  {-'s',-'a',-'R',1,0,1} },
  /*11*/ { (dec*)immsex,  {-'i',RT,-'r',3}, {-'i',-'R',0,SSAV+1} },
  /*12*/ { (dec*)immageX, {-'i',RT,-'s',2,-'r',3,-'a',6}, {-'i',-'r',0,SSAV+1,-'s',-'a'} },
  /*13*/ { (dec*)immage,  {-'i',RT,-'s',2,-'r',3,-'a',7}, {-'i',-'R',0,SSAV+1,-'A',-'s'} },
  /*14*/ { (dec*)M1,      {-'i',BY, -'s',2,-'a',AC}, {-'s',-'i',-'A'} },
  /*15*/ { (dec*)cft,     {-'a',121,-'d',2,-'i',RT} },
  /*16*/ { (dec*)d1,      {-'s',2,-'r',3,-'a',121}, {-'s',-'r',0,1,-'A'} },
  /*17*/ { (dec*)d2,      {-'s',2,-'r',3,-'a',124}, {-'s',-'r',0,2,-'A'} },
  /*18*/ { (dec*)d3,      {-'s',2,-'r',3,-'a',121}, {-'s',-'r',0,1,-'A'} },
  /*19*/ { (dec*)inf1981, {-'a',121,-'s',2,-'r',3,-'q',9}, {-'r',-'s',-'A',120,0,-'Q',1,8} },
  /*20*/ { (dec*)smear,   {-'a',121} },
  /*21*/ { (dec*)N3,      {-'a',4,  -'s',2,-'r',2,-'i',RT} },
  /*22*/ { (dec*)repc,    {-'a',4, -'s',2, -'r',2, -'d',2, -'i', RT} },
  /*23*/ { (dec*)sdimm,   {-'i',IS0+1} },
  /*24*/ { (dec*)sduk,    {-'i',IS1+1} },
         { } };
```

1.2 Main initialization

This routine should be called each time the program is reused, to clear static variables for the next run. The function was added when `tb30i.c` was made into a function of the fitting routine, to implement parallel, replicate runs of the tuberculosis program. This would not be necessary if the program were called as independent executable, as before.

```
MainInit()
{ int i,j,k,l,m,n;

  for(i=0; i<PN; i++) N[i] = 0;
  for(i=0; i<2;  i++) age1[i] = age2[i] = agec[i] = 0;

  for(i=0; i<4;  i++)
  for(j=0; j<2;  j++)
  for(k=0; k<3;  k++)
  for(l=0; l<2;  l++)
  for(m=0; m<RT; m++)
  for(n=0; n<5;  n++)
  { repc[i][j][k][l][m]     = 0;
    repc2[i][j][k][l][m][n] = 0;
    ari[j][m]               = 0;
    N2[i][j][k][m]          = 0;
    N3[i][j][k][m]          = 0;
    Np[i][j][k][m]          = 0;
    clust[i][j][l][n]       = 0; }

  for(i=0; i<15000; i++)
  for(j=0; j<7; j++)
    repc3[i][j]=0;

  deaths = events = immid = ukbid = repid = stid = 0;
  t = pt = 0;
}
```

1.3 Main program

Due to an MPI bug not allowing `popen` and related routines to work, the tuberculosis program is defined as a function which returns an array of output (rather than stand-alone executable) for use with the fitting routine. `define` statement is used to control whether tuberculosis program is a stand-alone executable or function within the fitting routine.

```
define main dec *mainiac
```
Make main not the real main, for use with fitting routine. Comment out for independent executable.

```
ifdef  main
static int fit5i = 1;
else
static int fit5i = 0;
endif
static int fitm  = 2;
```
Flag set when linked with fitting.

Flag set when not linked.

Flag set when fitting to rates, (0=numbers, 1=rates, 2=clust and 3=overall rates).

```
dec  out[1000]; int outi;
dec outn[1000]; int outni;
dec outc[1000]; int outci;
dec outo[1000]; int outoi;
```
Main output array (all).
Main output array (case numbers).
Main output array (clust).
Main output array (overall rates).

```
main(int argc, char *argv[])
{ int i, j, k, l, n, sid;

  startsec = time(NULL);
```
Retrieve the wall-clock time.

```
  if(fit5i==0) ErrorInit();
  MainInit();
  EventInit();
  FinalInit();
  ReportInit();
```
Trap system failures.
Start the main program.
Start the event queue.
Start the final reports.
Start the output reports.

```
  A = (struct Indiv *)
      calloc(indiv+3, sizeof(struct Indiv));
  if(A==0) Error(911.);
```
Allocate array of individuals. (Not static because of gcc bug restricting such arrays to 2GB.)

```
  if(SUPER) maximm = 900000;
  else      maximm = 500000;
```
Adjust `maximm` if not running on supercomputer.

```
  Data();
```
Read in appropriate data files and parameters, store to arrays.

```
  gparam(argc, argv);
```
Collect parameters which have been changed on the command line.

```
  Param();
```
Update parameters/distributions affected by parameters changed.

```
  if(bcy[0]<=0.0001)
  { ypb = RT*100;
    printf("Births are zero!\n"); }
  else ypb = 1./bcy[0];
  if(immig[0]<=0.0001)
  { ypi = RT*100;
    printf("Immigrants are zero!\n"); }
  else ypi = 1./immig[0];
```
Calculate years per birth and years per immigrant at t=t0 for scheduling. If none should occur, make interval very large, so they never happen.

```
  lup = t0;
```
Update time of last update for parameters sensitive to

```
                                                    calendar year.
  rand0 = abs(randseq);                             Start the random number sequence
  if(randseq>=0)  RandStart(rand0);                 from a specified or an arbitrary
  else rand0 = RandStartArb(rand0);                 place.

  EventStartTime(t0);                               Initialize the event queues.

  t = t0;                                           Set the starting time.

  stid = is0+is1;                                   Set first available new strain
                                                    strain ID for mutants.

  InitPop();                                        Set up initial population.

  Report(argv[0]); pt = t;                          Report initial conditions.

  BirthG();                                         Start external event generators
  ImmigrateG();                                     for birth and immigration.

  for(t=t0; t<t1; Dispatch())                       Main loop: process events,
  { if(t-pt<tgap) continue;                         reporting results periodically.
    pt = t; Report(argv[0]); }

  Report(argv[0]);                                  Get final report.
  Clust();                                          Get clustering statistics.
  Final();                                          Close processing and return to
  free(A);                                          caller.

  if(fit5i)                                         If linked with fitting routine,
    return fitm==0?outn: fitm==1?out:               return selected results.
           fitm==2?outc: outo;

  return 0;                                         Otherwise return a success code.
}
```

1.4 Dispatch next event

All events pass through this routine. It picks the earliest event in the list of pending events, sets the time to match that event, and performs the operations called for by that event. Typically that will result in other events being scheduled, to be seen in the future as they arrive at the top of the list of pending events.

Entry: The system is initialized with all events in the list ready for processing.
`t` contains the present time.
`t1` contains the ending time.

Exit: The next event has been processed and `events` incremented, if the events `time is less than` t1'.
`t` is advanced to the next event, which may be an unprocessed event at time greater than `t1`.

```
Dispatch()
{ int n; dec tw;
  tw = t;                                           Remember the previous time.
  n = EventNext(); if(t>t1) return;                 Advance time to the next event.
  tstep(tw, t);                                     Record the size of the time step.
  events += 1;                                      Increment the events counter.

  switch(A[n].pending)                              Process the event.
  { case pVaccin:   Vaccination(n);  break;         [Vaccination]
```

```
    case pTransm:   Transmission(n); break;  [Transmission of an infection]
    case pRemote:   Remote(n);       break;  [Transition to latency]
    case pDisease:  Disease(n);      break;  [Progression to disease]
    case pDeath:    Death(n);        break;  [Death]
    case pMutate:   Mutate(n);       break;  [Strain type mutation]
    case pEmigrate: Emigrate(n);     break;  [Emigration]
    case pBirth:    BirthG();        break;  [Birth generator]
    case pImmig:    ImmigrateG();    break;  [Immigration generator]
    case pRep:      Rep(n);          break;  [Case report]

    default: Error2(921.2,                    [System error]
      "`A[",n,"].pending=",A[n].pending); }
}
```

1.5 Birth

This routine is dispatched when an individual is to be born. All newborns are Uninfected; exit from the Uninfected compartment is by vaccination to the Immune compartment, by infection to the Recent Infection compartment, and by emigration from the population or death.

Entry: `n` indexes an individual being born.
`b` contains the time of birth. Presently, this is the current time, though with some set up, it could be earlier than present (notably, `pmale` would have to be indexed differently).
`t` contains the current time.
`A[n].state` contains the present state of the record (can be any state, including 0, which is not a state but marks records not yet assigned).
`m1` contains the mortality rate for susceptible individuals (if app.).
`em` contains the emigration rate.
`v1` contains vaccine efficacy.
`v2` contains the probability that an individual will be vaccinated.
`v3` contains the average age of vaccination.
`VTYPE` is zero if vaccinations are to match ODE conventions.
No event is scheduled for individual `n`.

Exit: `Birth` contains a status code.
0 The individual would die before the current time so no birth has been recorded and no event scheduled.
1 Entry `n` is initialized as a susceptible newborn and its first event is scheduled, either vaccination, emigration or death.
`A[n].state` marks a susceptible individual.
Counters in `N` are updated.

```
define VTYPE 1                                  Vaccination type.

int Birth(int n, dec b)
{ int y, s, v, e; dec wd, we, wv;
  if(n<maximm+1) Error1(610.1, "n=",(dec)n);    Check for appropriate n, this
  if(n>indiv)    Error1(610.2, "n=",(dec)n);    routine does not allow immigrant
                                                births or births to those with
                                                index number greater than indiv.
  y = (int)t - (int)t0;                         Retrieve year index for arrays.
  A[n].sex = Rand()<pmale[y]? 0: 1;             Assign the newborn's sex.
  s = A[n].sex;
  wd = b+LifeDsn(s,t-b,m1[s][y]);               Schedule a time of death and
  if(wd<t) Error(850.);                         check for errors.
```

```
  we = b+EmDsn(1,s,t-b,em[s][UK]);
```
Calculate time of emigration.

```
  A[n].tBirth    = b;
  A[n].tDeath    = wd;
  A[n].tEmigrate = we;
  A[n].rob       = 1;
  NewState(n, qU);
  A[n].tExit     = 0;
  A[n].tDisease  = 0;
  A[n].tTransm   = 0;
  A[n].tMutate   = 0;
  if(REC) A[n].inf = 1;
  if(SSAV) A[n].ssa = 0;
  A[n].strain = 0;
```
Record the time of birth.
Record the time of death.
Record the time of emigration.
Set as born in UK.
Mark as Uninfected.
Clear any other saved event times or states.

```
  v = 0; switch(VTYPE)
  {
  case 0:
    wv = b+Expon(v1[UK]*v2[UK]/v3[UK]);
    if(wv<wd && wv<we) v = 1;
    break;
```
Select the type of vaccination scheduling.

Generate a time for vaccination compatible with ODE models (for testing).

```
  case 1:
    wv = b+v3[UK]+Rand();
    if(b<1993 && Rand()<(v1[UK]*v2[UK])
              && wv<wd && wv<we) v = 1;
    break;
```
Generate a vaccination sometime within the specified year if probabilities allow.

```
  default: Error1(611., "",(dec)VTYPE);
  }
```
Improper vaccination type.

```
  if(v)
  { A[n].pending = pVaccin;
    EventSchedule(n, wv);
    return 1; }
```
If vaccination occurs before death and emigration, schedule the vaccination.

```
  if(we<wd)
  { A[n].pending = pEmigrate;
    EventSchedule(n, we);
    return 1; }
```
Schedule emigration if that is the earliest event.

```
  { A[n].pending = pDeath;
    EventSchedule(n, wd);
    return 1; }
}
```
Otherwise, schedule death.

1.6 Immigration

This routine brings a new individual into the population from outside. The new individual assigned demographic and infection-related attributes according to appropriate probability distributions. They are then scheduled for their earliest event. All information stored for this individual is written over, in case their index number is being recycled from an individual leaving the study population through death or emigration.

Entry: `n` contains the index number of the new immigrant. The contents of the record is undefined.
`t` contains the current time.
`indiv` contains the highest index number for any individual.
`maximm` contains the highest index number for an immigrant.

`t0` contains begining time of the simulation.
`SSAV` contains model version, 0=non-SSA, 1=SSA
`ssaim` contains proportion of non-UK born immigrants from SSA by year.
`immsex[r]` contains the proportion of immigrants who are male, r=0, non-UK born; r=1, UK-born; r=2, SSA-born. Note in SSA and non-SSA versions of the model, `non-UK born` will be defined differently.
`hivp` contains the HIV prevalence, by sex and year, for SSA immigrants.
`infimm` contains cumulative probabilities immigrants enter disease states, by age, rob and calendar year.
`Ax` contains state variables which accompany `infimm`.
`M1` contains the mortality table for non-diseased (real runs only).
`A1` contains state variables which accompany `M1`.
`m1` contains the mortality rate (ODE validation only).
`em` contains the emigration rate.
`v1` contains vaccine efficacy.
`v2` contains the probability that an individual will be vaccinated.
`v3` contains the average age of vaccination.
No event is scheduled for individual `n`.

Exit: An event is scheduled for individual `n`.
`A[n].state` contains the disease state.
`A[n].tEntry` contains the time of entry to the new state.
`A[n].tBirth` contains the time of birth.
`A[n].tImm` contains the time of immigration.
`A[n].tDeath` contains the time of death.
`A[n].tEmigrate` contains the time of emigration.
`A[n].sex` contains the sex.

```
int Immigrate(int n)
{ int y,s,rob,rob2,ac,a,st; dec r,age,wd,we,wv,tinf;
  if(n>indiv) Error1(610.3, "n=",(dec)n);     Check for appropriate n.
  if(n<1)     Error1(610.4, "n=",(dec)n);
```

Set up basic, uninfected immigrant

```
  NewState(n,qU);                              Assign to Uninfected state
                                               to start with.
  y = (int)t - (int)t0;                        Get array index for year.

  A[n].tExit     = 0;                          Clear saved event
  A[n].tDisease  = 0;                          times or states.
  A[n].tTransm   = 0;
  A[n].tMutate   = 0;
  if(REC) A[n].inf = 1;
  if(SSAV) A[n].ssa = 0;
  A[n].strain = 0;
  if(n<=maximm) A[n].rob = rob = 0;            Assign rob=0 to all non-UK born.
  else          A[n].rob = rob = 1;            Assign rob=1 to UK-born.

  s = 0;                                       Set sex as male to begin.

  if(rob==0 && SSAV==1)                        If non-UK born and running
  { A[n].ssa = 0;                              SSA version of model, check
    if(Rand()<ssaim[y])                        to see if SSA. If so, get their
    { A[n].ssa = 1;                            sex and HIV status.
      if(Rand()>immsex[y][SSA]) s = 1;         Get sex of SSA.
      if(Rand()<hivp[s][y]) A[n].ssa = 2; }    Get HIV status of SSA.
    else                                       If not SSA, leave as other non-
      if(Rand()>immsex[y][0]) s = 1;           UK and assign sex.
```

```
}
```

```
else                                          Assign sex to non-UK-born in
  if(Rand()>immsex[y][rob]) s = 1;            non-SSAV model and UK-born.
```

```
A[n].sex = s;                                 Assign sex to record.
```

```
age=GetAge(n,s,rob);                          Assign age.
a = (int) age;                                Save integer age.
rob2=rob;                                     Create rob2 (0,1 and 2)
if(SSAV && A[n].ssa) rob2=2;                  since here rob is only 0 or 1).
```

```
A[n].tBirth = t-age;                          Save time of birth based on age.
```

```
A[n].tDeath = wd                              Assign time of death and check
            = t+LifeDsn(s,age,m1[s][y]);      death time is ok.
if(wd<A[n].tBirth+age) Error(612.1);
```

```
A[n].tEmigrate                                Assign emigration time.
     = we
     = t+EmDsn(rob2,s,age,em[s][rob2]);
if(age<v3[rob] && Rand()<v1[rob]*v2[rob]      Determine if vaccination should
               && t<2005-(v3[rob]-age))       occur and assign vaccination
  wv = t+(v3[rob]-age)+Rand();                time if so. If it should not
else wv = t+2*RT+Rand();                      occur, set to time which will
                                              not happen in the model.
if(wv<wd && wv<we)                            Schedule vaccination if it is
{ A[n].pending = pVaccin;                     the earliest event.
  EventSchedule(n, wv); }
```

```
else if(wd<we)                                Schedule death if it is the
{ A[n].tExit = wd;                            earliest event.
  A[n].pending = pDeath;
  EventSchedule(n, wd); }
```

```
else                                          Otherwise schedule emigration if
{ A[n].tExit = we;                            it is the earliest event.
  A[n].pending = pEmigrate;
  EventSchedule(n, we); }
```

```
A[n].tDisease  = 0;                           Clear time to disease.
A[n].tTransm   = 0;                           Clear time to transmit.
A[n].tMutate   = 0;                           Clear time of strain mutation.
```

Assign disease state to immigrant and process accordingly

```
st = 1+
  (int)RandF(Ax,infimm[a][rob2][y],9,1.0);    Get random disease state.
if(st==1)                                     Do nothing if Uninfected.
  return 0;
```

```
else if(st==2)                                Process Immune.
{ EventCancel(n);
  Vaccination(n);
  return 1; }
```

```
else if(st==3)                                Process Recently Infected.
{ tinf = Rand()*5;                            Set random time since infection
  Infect(n,tinf,Strain(0),0);                 within the last five years.
  if(REC) A[n].inf = 3;                       Assign inf (time/place).
  return 2; }
```

```
else if(st==4)                                Process Remotely infected.
```

```
  { EventCancel(n);
    A[n].strain = Strain(0);                Assign infection strain.
    NewState(n,qD1);                       Put in temporary disease state
    Remote(n);                             to facilitate re-scheduling of
                                           events in Remote() function.
    if(REC) A[n].inf = 4;                  Assign inf (time/place).
    return 3; }

  else if(st==5)                           Process Reinfected.
  { NewState(n,qI2);                       Put in Remote infection so
    tinf = Rand()*5;                       that Infect() picks this up
    Infect(n,tinf,Strain(0),0);            as reinfection, also draw
                                           random infection time.
    if(REC) A[n].inf = 3;                  Assign inf (time/place).
    return 4; }

  else if(st>5 && st<9)                    Process Primary, Reactivation,
  { EventCancel(n);                        Reinfection disease classes.
    A[n].strain = Strain(0);               Assign infection strain.
    NewState(n,st-3);                      First put in correct infection
    Disease(n);                            class and then send to Disease.
    if(REC)                                Assign inf (time/place)
    { if(st==7) A[n].inf = 4;
      else      A[n].inf = 3; }
    return 5; }

  else
    Error(618.1);
    return 0;                              (Will never reach this).
}
```

1.7 Vaccination

This routine is dispatched when an individual is scheduled for an effective vaccination. Ineffective vaccinations are already accounted for - they are never scheduled for this routine. Effective vaccinations are assumed to impart lifelong immunity; therefore individuals will never leave this state, except by dying or emigrating from the population.

Entry: `n` indexes an individual being born.
`t` contains the current time.
`A[n].state` contains the present state (always `qU`).
`A[n].sex` contains the individual's sex.
`A[n].tEmigrate` contains time of emigration.
`A[n].tDeath` contains time of death.
No event is scheduled for individual `n`.

Exit: `A[n].state` contains the new state, qV.
Counters in `N` are updated.

```
int Vaccination(int n)
{
  NewState(n, qV);                         Change states.

  if(A[n].tEmigrate<A[n].tDeath)           Schedule emigration if that is
  { A[n].pending = pEmigrate;              the earliest event.
    EventSchedule(n, A[n].tEmigrate); }
  else                                     Otherwise, schedule death.
  { A[n].pending = pDeath;
```

```
    EventSchedule(n, A[n].tDeath); }
  return 0;
}
```

1.8 Infect a specified individual

Individuals receive infections from others via this routine. If the targeted individual is in the Uninfected or Remote Infection state, it acquires the infection and moves to the Recent Infection or Reinfection state. The individual is then scheduled to return to Remote Infection, develop disease, emigrate, die or have its strain type mutate, depending on probabilities of each and random chance. If the individual targeted for infection is in one of the other nine disease states, they are not susceptible to infection and the transmission event has no effect.

Entry: `n` indexes the individual to be infected.
`tinf` contains the time of infection, ¡=5 years ago.
`strain` contains the strain ID number of infecting strain.
`type` differentiates infections at initialization or migration,
type=0 or transmission during the simulation, type=1.
`t` contains the current time.
`A[n].state` contains the state of infection target.
`A[n].tEmigrate` contains the time of emigration of infection target.
`A[n].tDeath` contains the time of death for infection target.
`r1` and `r2` contain the latency rates for `qI1` and `qI3`.
`mi` contains the mutation rate for infection strains.
An event is still scheduled for individual `n`.

Exit: `Infect` describes the result.
0 The specified individual could not be infected or reinfected.
1 Return to remote infection is scheduled.
2 Disease is scheduled.
3 Death is scheduled.
4 Strain mutation is scheduled.
5 Emigration is scheduled.
`A[n].state` contains the new state, if applicable.
Entry `n` is infected (if `Infect` is nonzero).
Counters in `N` are updated.

```
int Infect(int n, dec tinf, int strain, int type)
{ int a,s,rob,y,q; dec d,r,wd,we,wdis,wr,wm;
  if(n>indiv||n<1)   Error1(610.3,"",n);      Check for appropriate n.
  if(strain>=stid)   Error1(616.0,"",strain);Check for appropriate strain ID.
  if(tinf>5||tinf<0) Error1(617.0,"",tinf);  Check for appropriate tinf.
  if(tinf==5) tinf=tinf-E;                    Correct tinf if equal to 5.

  a   = (int)(t-A[n].tBirth);                 Retrieve integer age.
  s   = A[n].sex;                             Retrieve sex.
  rob = A[n].rob;                             Retrieve region of birth.
  y   = (int)t - (int)t0;                     Get array index for year.

  switch(A[n].state)                          Determine the new state and
  {                                           its associated parameters.
    case qI2: r=r2[s]; q=qI3; break;
    case qU:  r=r1[s]; q=qI1; break;
    default:  return 0;                       Avoid uninfectable states.
  }
```

```
  EventCancel(n);                          Else cancel the pending event and
  NewState(n, q);                          mark this individual as infected.
  if(REC) A[n].inf  = 1;                   Save place of infection as UK
                                           (will be changed outside routine
                                           if infection is acquired abroad).
  if(type==1)                              Increment ARI counter if infection
   ari[rob][y]++;                          occurred in the UK.

  A[n].strain = strain;                    Assign infecting strain type.

  wd = A[n].tDeath;                        Retrieve time of death.
  we = A[n].tEmigrate;                     Retrieve time of emigration.
  wr = t+LAT-tinf;                         After LAT years individual is
                                           defined as remotely infected.
  wdis = t+Tdis(n,a,s,rob,tinf)+E;         Calculate time to disease.
  if(wdis<=t) Error2(620.0,"t=",t, " wdis=",wdis);
  wm   = t+Expon(mi);                      Calculate strain mutation time.

  if(wd<we && wd<wr && wd<wdis && wd<wm)   If death is earliest event,
  { A[n].pending = pDeath;                 schedule the death and
    EventSchedule(n, wd);                  ignore everything else.
    return 3; }

  if(we<wr && we<wdis && we<wm)            If emigration is the earliest
  { A[n].pending = pEmigrate;              event, schedule it and
    EventSchedule(n, we);                  ignore everything else.
    return 5; }

  if(wr<wdis && wr<wm)                     Otherwise, if transition to
  { A[n].pending = pRemote;                remote infection is the
    EventSchedule(n, wr);                  schedule that, save mutation
    A[n].tMutate = wm;                     time, and ignore disease
    return 1; }                            time.

  if(wm<wdis)                              Otherwise, if mutation is the
  { A[n].pending = pMutate;                earliest event, schedule
    EventSchedule(n, wm);                  that and save time to disease
    A[n].tDisease = wdis;                  and time to remote.
    A[n].tExit = wr;
    return 4; }

  { A[n].pending = pDisease;               Otherwise, schedule disease and
    EventSchedule(n, wdis);                ignore other times, as they will
    return 2; }                            be recalculated at disease onset.
}
```

1.9 Enter compartment remote

This routine is dispatched when an infection becomes latent, entering the Latent Infection state (qI2). Recent Infection (qI1), Reinfection (qI3), and all disease compartments (qD1-qD6) can lead to this state. The Latent Infection state allows the possibility of progression to disease, mutation of strain type, death and emigration. Also, those with Latent Infection can be reinfected, although that is induced by a transmission event dispatched independently and not handled here.

Entry: n indexes the individual.
t contains the current time.
A[n].state contains the present state (can be qI1, qI3, qD1-qD6).

A[n].tMutate contains the strain mutation time.
d2 contains the disease progression rate for qI2.
m4 contains the mortality rate for qI2.
mi contains the mutation rate of strain types in those infected, but not diseased.
No event is scheduled for individual n.

Exit: Remote contains a status code:
2 Disease is scheduled.
3 Death is scheduled.
4 Mutation is scheduled.
5 Emigration is scheduled.
A[n].state represents Remote Infection (qI2).
A[n].tDeath is updated as necessary.
A[n].tMutate is updated as necessary.
Counters in N are updated.

```
int Remote(int n)
{ int y, a, s, rob, q; dec age, wdis, wd, we, wm;
  y   = (int)t - (int)t0;                    Retrieve array index for year.
  age = t-A[n].tBirth;                       Retrieve age.
  a   = (int) age;                           Integer age.
  s   = A[n].sex;                            Retrieve sex.
  rob = A[n].rob;                            Retrieve region of birth.

  q = A[n].state;                            Remember the previous state.
  NewState(n, qI2);                          Mark the individual as remote.
  if(REC)                                    Adjust inf to older infection
    if(A[n].inf==1 || A[n].inf==3)           if not already older.
      A[n].inf+=1;
  if(q>=qD1)                                 Establish a new time for strain
     A[n].tMutate = t+Expon(mi);             mutation if the prior state
                                             was disease.
  wdis = t+Tdis(n,a,s,rob,0);                Calculate time to disease.
  wd = A[n].tDeath;                          Retrieve time of death.
  we = A[n].tEmigrate;                       Retrieve time of emigration.
  wm = A[n].tMutate;                         Retrieve time of mutation.
  if(wd<wdis && wd<wm && wd<we)              If death is the earliest event,
  { A[n].pending = pDeath;                   schedule it.
    EventSchedule(n, wd);
    return 3; }

  if(wm<wdis && wm<we)                       Otherwise, if strain mutation
  { A[n].pending = pMutate;                  is the earliest event,
    EventSchedule(n, wm);                    schedule strain it and save
    A[n].tDisease = wdis;                    time to disease onset.
    return 4; }

  if(we<wdis)                                Otherwise, if emigration is
  { A[n].pending = pEmigrate;                the earliest event, schedule
    EventSchedule(n,we);                     it and ignore other times.
    return 5; }

  { A[n].pending = pDisease;                 Otherwise, schedule progression
    EventSchedule(n, wdis);                  to disease and ignore
    return 2; }                              everything else.
}
```

1.10 Disease

This routine is dispatched when an infection progresses to active disease. There are six distinct disease compartments, pulmonary and non-pulmonary compartments for Primary (qD1/qD4), Reactivation (qD2/qD5), and Reinfection (qD3/qD6). Individuals enter disease compartments from three infection compartments – Recent Infection, Latent Infection, and Reinfection – which determine the disease compartment they enter. This routine handles all transitions to disease.

Future events for diseased individuals include transmission of infection to others, recovery to Latent Infection, death, emigration, reporting of their disease case, or strain type mutation.

Entry: `n` indexes the individual progressing to disease.
`t` contains the current time.
`A[n].state` contains the state progressing to disease (can be `qI1`, `qI2`, or `qI3`).
`A[n].tBirth` contains the time of birth of the individual.
`A[n].sex` contains the sex of the individual.
`A[n].rob` contains region of birth of the individual.
`A[n].tEmigrate` contains the time of emigration.
`A[n].tDeath` contains the scheduled time of death.
`cft` contains the case fatality rate (actually a proportion).
`r3`, `r4`, `r5`,`r6`,`r7`, and `r8` contain recovery rates for `qD1`, `qD2`, `qD3`, `qD4`, `qD5`, and `qD6`, respectively.
`m6`, `m7`, `m8`, `m9`,`m10`, and `m11` contain mortality rates for the states named above in the ODE-compatible version of the model.
`p1`, `p2`, and `p3` contain the fraction of disease which becomes pulmonary, from sources `I1`, `I2`, and `I3`, respectively.
`c` contains the average number of new infections produced by this individual per year.
`proprep` contains the proportion of cases reported.
`md` contains the mutation rate for strains involved in disease.
No event is scheduled for individual `n`.

Exit: `Disease` contains a status code.
1 A transmission is scheduled.
2 Recovery is scheduled.
3 Death is scheduled.
4 Strain mutation is scheduled.
5 Emigration is scheduled.
6 Case report is scheduled.
`A[n].state` contains the new state.
`A[n].tDeath` contains a possibly updated time of death.
`A[n].tMutate` contains the new time of strain mutation, if applicable.
`A[n].tTransm` contains the next time of transmission, if applicable.
`A[n].tExit` contains the time of recovery to remote infection, or if would happen after death, the time of death.
`A[n].tRep` contains the time of disease case report, if applicable.
Counters in `N` are updated.

```
int Disease(int n)
{ int a, s, rob, y, ds, q; dec age, r, m, p, wm, we, wd, wr, wt, e, wrep;
  age = t-A[n].tBirth;                    Retrieve age.
  a   = (int)age;                         Calculate integer age.
  s   = A[n].sex;                         Retrieve sex.
  rob = A[n].rob;                         Retrieve region of birth, 0 or 1.
```

```
y   = (int)t - (int)t0;
```
Retrieve array index for year.

```
switch(A[n].state)
{
  case qI1: r=r3[s]; m=m6[s][y]; p=p1[a][s][rob]; q=qD1; break;
  case qI2: r=r4[s]; m=m7[s][y]; p=p2[a][s][rob]; q=qD2; break;
  case qI3: r=r5[s]; m=m8[s][y]; p=p3[a][s][rob]; q=qD3; break;
  default:  Error(922.0); }
```
Determine the new state and its associated parameters.

```
if(Rand()>p)
{ switch(A[n].state)
  {
    case qI1: r=r6[s]; m= m9[s][y]; q=qD4; break;
    case qI2: r=r7[s]; m=m10[s][y]; q=qD5; break;
    case qI3: r=r8[s]; m=m11[s][y]; q=qD6; break;
    default:  Error(922.0); } }
```
If this should be non-pulmonary disease, update new state and associated parameters.

```
NewState(n, q);
```
Mark the individual as diseased.

```
wr = A[n].tExit = t+RecovDsn(s,age,r);
```
Establish time to remote.

```
we = A[n].tEmigrate;
```
Retrieve emigration time.

```
wd = A[n].tDeath;
```
Retrieve time of death.

```
A[n].tMutate = wm = t+Expon(md);
```
Establish new mutation time.

```
if(q>=qD4) ds = 0;
else       ds = 1;
```
Set disease site to non-pulmonary or pulmonary.

```
if(Rand()<cft[a][ds][y])
{ if(wr<wd && wr<we) e = wr;
  else if(wd<we)     e = wd;
  else               e = we;

  wd = t+0.99*(e-t);

  A[n].tDeath = wd; }
```
If person should die from disease, find earliest of death, emigration and recovery to assign death before these occur, assigning death time close to the end of disease duration.

Since disease death is before natural death time, replace it.

```
if(Rand()<proprep)
{ if(wr<wd && wr<we) e = wr;

  else if(wd<we)     e = wd;

  else               e = we;

  A[n].tRep = t+Rand()*(e-t); }
```
If case should be reported, find reporting time.
First find earliest of death, emigration and disease recovery to get range for reporting time.

Randomly assign reporting time.

```
else
  A[n].tRep = t+2*RT+Rand();
```
If case should not be reported, assign a reporting time beyond running time of model.

```
if(A[n].tRep==0) Error1(619., "n=",n);
wrep = A[n].tRep;
```
Save reporting time.

```
if(wd<wr)
  wr = wd;
```
If death would occur before recovery, give death precedence.

```
if(q<qD4 && Rand()<smear[a])
  wt = t+Expon(c[s][rob]);
else wt = t+2*RT + Rand();
A[n].tTransm = wt;
```
If this is pulmonary disease and smear positive, store time to transmit. Otherwise, set time to transmit which never happens.

```
if(wt<wr && wt<wm && wt<we && wt<wrep)
{ A[n].pending = pTransm;
  EventSchedule(n, wt);
```
If transmission is earliest event, schedule it.

```
    return 1; }
  if(wrep<wr && wrep<wm && wrep<we)
  { A[n].pending = pRep;
    EventSchedule(n, wrep);
    return 6; }
  if(wr<wd && wr<wm && wr<we)
  { A[n].pending = pRemote;
    EventSchedule(n, wr);
    return 2; }
  if(wm<wd && wm<we)
  { A[n].pending = pMutate;
    EventSchedule(n, wm);
    return 4; }
  if(we<wd)
  { A[n].pending = pEmigrate;
    EventSchedule(n,we);
    return 5; }
  { A[n].pending = pDeath;
    EventSchedule(n, wd);
    return 3; }
}
```

If case report is earliest event, schedule it.

If recovery is the earliest event, schedule it.

If mutation is the earliest event, schedule it.

If emigration is the earliest event, schedule it.

Otherwise schedule death.

1.11 Transmission

Infectious individuals transmit infection via this routine. Another individual is selected to be infected, either randomly from within the same region of birth as the infectious individual (UK-born or non-UK born, even in SSA version of model), or randomly from the entire population. If the target individual is susceptible (Uninfected or Remote Infection states), infection is established and that individual is processed accordingly. If not, no transmission occurs.

After the transmission attempt, the infectious individual is then scheduled for another transmission, strain type mutation, recovery, case report, emigration or death.

Entry: `n` indexes the individual to transmit an infection.
`t` contains the current time.
`A[n].tExit` contains the time of recovery, or if that time is equal to or greater than the time of death, contains time of death.
`A[n].tDeath` contains the time of death.
`A[n].tMutate` contains the strain type mutation time.
`A[n].tEmigrate` contains the emigration time.
`A[n].rob` contains the region of birth, UK or non-UK.
`A[n].sex` contains the sex.
`A[n].strain` contains the infection strain ID number.
`pcc` contains the proportion of close contacts.
`maximm` contains the maximum ID number for immigrants.
Non-UK born individuals are indexed from 1 to (immid-1), a total of (immid-1) individuals.
UK-born individuals are indexed from (maximm+1) to (ukbid-1), a total of (ukbid-maximm-1) individuals.
No event is scheduled for individual `n`.'

Exit: A new individual has been targeted for infection. If the infection takes hold, that individual is scheduled for strain mutation, disease,

remote infection, emigration or death.

`Transmission` contains a status code.

1 Another infection is scheduled.
2 Recovery to remote infection is scheduled.
3 Death is scheduled.
4 Strain mutation is scheduled.
5 Emigration is scheduled.
6 Case report is scheduled.

Counters in `N` are updated.

```
define SCHED(X,Y,Z)  { A[n].pending = X; EventSchedule(n,Y); return Z; }

int Transmission(int n)
{ int i, j, y, low, tot; dec age;
  static int v[] = { iTransm,iDeath,iEmigrate,iExit,iMutate,iRep, -1 };
  y = (int)t - (int)t0;                          Get year for array index.

  if(Rand()<pcc)                                 If targetting close contact
  { if(A[n].rob)                                 obtain total number and lowest
    { low  = maximm+1;                           ID number from individual's own
      tot = (ukbid-1) - low + 1; }               region of birth, for selection
    else                                         of random target for infection
      { low  = 1;                                within that region of birth.
      tot = immid-1 - low + 1; }

    do i=low+(int)(Rand()*tot);                  Find person other than self to
    while(i==n);                                 infect.
  }

  else                                           If not a close contact, choose
  { do                                           random person to infect from
    { tot = (immid-1) + (ukbid-maximm-1);        entire population, increment
      j = 1 + (int)(Rand()*tot);                 Adjust ID numbers for UK-born.
      if(j>=immid)
        i = j+(maximm+1-immid);
      else
        i = j;  }
    while (i==n);                                Avoid infecting self.
  }
  Infect(i,0,A[n].strain,1);                     Infect chosen individual.

  A[n].tTransm=t+Expon(c[A[n].sex][A[n].rob]);Establish time to transmit again.

  switch(i=Earliest(A[n].t, v))                  Schedule the earliest event.
  { case iRep:      SCHED(pRep,      A[n].tRep,      6);
    case iTransm:   SCHED(pTransm,   A[n].tTransm,   1);
    case iExit:     SCHED(pRemote,   A[n].tExit,     2);
    case iMutate:   SCHED(pMutate,   A[n].tMutate,   4);
    case iEmigrate: SCHED(pEmigrate, A[n].tEmigrate, 5);
    case iDeath:    SCHED(pDeath,    A[n].tDeath,    3);
    default:        Error1(922., "m=",(dec)i);           }

  return 0;                                      (Will never reach this.)
}
```

1.12 Mutation

This routine is dispatched when the strain type of an infected or diseased individual is scheduled for mutation. The mutation does not affect any other event. After mutation is processed, the individual is scheduled for their next event, which will depend on disease state.

Entry: `n` indexes an individual whose strain type is to mutate
`t` contains the current time.
`mi` contains the mutation rate for strains not in an active disease case (merely infection).
`md` contains the mutation rate for strains in a disease case.
`A[n].state` contains the current state (can be any infection or disease state).
`A[n].strain` contains the strain identification number of the current strain of infection or disease.
`A[n].tDeath` contains the saved time of death.
`A[n].tEmigrate` contains the saved time of emigration.
`A[n].tExit` contains the saved time to exit state.
`A[n].tTransm` contains the saved time to transmit again.
No event is scheduled for individual `n`.

Exit: The next event for individual `n` is scheduled.
`A[n].tMutate` contains the time of next scheduled strain mutation.
`Mutation` contains a status code.
1 Recovery to remote infection is scheduled.
2 Progression to disease is scheduled.
3 Death is scheduled.
4 Strain mutation is scheduled.
5 Emigration is scheduled.
6 Case report is scheduled.
Counters in `N` are updated.

```
Mutate(int n)
{ dec m, wm, wd, we, wdis, wr, wt, wrep;
  A[n].strain = stid++;                    Assign new, mutant strain type
                                           and update next available ID.

  if(A[n].state<=qI3) m = mi;              Determine appropriate mutation
  else m = md;                             rate and calculate time to
  wm = t+Expon(m);                         next mutation.

  wd   = A[n].tDeath;                      Get time of death.
  we   = A[n].tEmigrate;                   Get time of emigration.
  wdis = A[n].tDisease;                    Get time of disease.
  wr   = A[n].tExit;                       Get time to remote infection.

  if(A[n].state==qI2)                      Schedule events for remotely
  {                                        infected individuals (qI2).
    if(wd<we && wd<wdis && wd<wm)
    { A[n].pending = pDeath;               If death occurs first,
      EventSchedule(n, wd);                schedule it.
      return 3; }

    if(wm<we && wm<wdis)                   Otherwise, if strain mutation
    { A[n].pending = pMutate;              occurs first, schedule it.
      EventSchedule(n, wm);
      return 4; }
```

```
  if(wdis<we)                                  Otherwise, if disease occurs
  { A[n].pending = pDisease;                   first, schedule it.
    EventSchedule(n, wdis);
    return 2; }

  { A[n].pending = pEmigrate;                  Otherwise, schedule emigration.
    EventSchedule(n, we);
    return 5; }
}

if(A[n].state<=qI3)                            Schedule events for the other
{                                              infected classes (qI1, qI3).
  if(wd<wdis && wd<wr && wd<wm && wd<we)
  { A[n].pending = pDeath;                     If death is earliest event,
    EventSchedule(n, wd);                      schedule it.
    return 3; }

  if(wr<wdis && wr<wm && wr<we)                Otherwise, if transition to
  { A[n].pending = pRemote;                    remote infection occurs
    EventSchedule(n, wr);                      first, schedule it and
    A[n].tMutate = wm;                         save mutation time.
    return 1; }

  if(wm<wdis && wm<we)                         Otherwise, if mutation is the
  { A[n].pending = pMutate;                    earliest event, schedule it.
    EventSchedule(n, wm);
    return 4; }

  if(wdis<we)
  { A[n].pending = pDisease;                   Otherwise, if disease onset
    EventSchedule(n, wdis);                    is earliest, schedule it.
    return 2; }

  { A[n].pending = pEmigrate;                  Otherwise, schedule emigration.
    EventSchedule(n, we);
    return 5; }
}

{                                              Schedule events for diseased.
  wrep = A[n].tRep;                            Get time of case report.
  if(A[n].state<qD4)                           If this is pulmonary disease,
  { wt = A[n].tTransm;                         retrieve time for transmission
    if(wt<wd && wt<wr && wt<wm && wt<we &&      wt<wrep)
    { A[n].pending = pTransm;                  and if it is the earliest
      EventSchedule(n, wt);                    event, schedule it and
      A[n].tMutate = wm;                       save mutation time.
      return 1; } }

  if(wrep<wd && wrep<wr && wrep<wm && wrep<we)
  { A[n].pending = pRep;                       If case report is the earliest
    EventSchedule(n, wrep);                    event, schedule it and save
    A[n].tMutate = wm;                         mutation time.
    return 6; }

  if(wr<wd && wr<wm && wr<we)                  If recovery is the earliest
  { A[n].pending = pRemote;                    event, schedule it.
  EventSchedule(n, wr);
  return 2; }

  if(wm<wd && wm<we)                           If mutation is the earliest
  { A[n].pending = pMutate;                    event, schedule it.
```

```
    EventSchedule(n, wm);
    return 4; }

    if(wd<we)
    { A[n].pending = pDeath;                    If death is the earliest
    EventSchedule(n, wd);                       event, schedule it.
    return 3; }

    { A[n].pending = pEmigrate;                 Otherwise, schedule emigration.
      EventSchedule(n, we);
      return 5; }
  }
}
```

1.13 Death

This routine is dispatched when individuals die. The routine logs dead individuals out of compartments as they leave the study population, to keep track of numbers of individuals in each compartment so that the array of individuals does not have to be scanned for that information. Also, the individuals `index number is recycled with` Transfer`so that array` A' is always continuous.

Entry: `n` indexes an individual who has just died.
`t` contains the current time.
`A[n].state` contains the present state (can be any compartment).
`A[n].tBirth` contains the time of birth.
`ukbid` contains the next available index number for UK-born individuals.
No event is scheduled for individual `n`.

Exit: Either entry `n` is sent to the `Birth()` function, to be initialized as a susceptible newborn and function returns 0 (`DTYPE==0`) or index number is recycled with `Transfer` (`DTYPE==1`), no birth is generated and function returns 1.
`N[A[n].state]` is decremented.
Counters `age1`, `age2`, and `agec` are updated.
Counters in `N` are updated.
`deaths` is incremented.

```
define DTYPE 1                                  Allows for non-constant population
                                                size.
Death(int n)
{ int n2; dec age;
  deaths += 1;                                  Increment the number of deaths.
  N[A[n].state]-=1;                             Decrement the number in the state.
  age = t-A[n].tBirth;                          Compute the age at death.

  { age1[0] += age; age2[0] += age*age;         Accumulate statistics for mean
    agec[0] += 1; }                             age and its variance.

  if(DTYPE==0)                                  If population size is to be held
  { Birth(n, t);                                constant, initiate a birth.
    return 0; }

  if(A[n].rob)                                  Avoid unoccupied index numbers
  { n2 = ukbid-1; ukbid--; }                    in array A by transferring
                                                highest-numbered individual, n2,
  else                                          to index number n.
  { n2 = immid-1; immid--; }
```

```
  Transfer(n, n2);
  return 1;
}
```

1.14 Emigration

This routine is dispatched when individuals migrate out of the study population. This routine logs the individual out of its compartment as they leave the population, maintaining numbers in each compartment so that the array of individuals never has to be scanned for that information. Also, the individual's index number is recycled with `Transfer` so that array `A` is always continuous.

Entry: `n` indexes the individual.
`tli` contains the time of last immigration.
`A[n].state` contains the disease state.
`A[n].rob` contains the region of birth.
`immig[y]` contains the total number of immigrants each year.

Exit: `N[A[n].state]` is decremented.
`n` is recycled such that the highest-numbered individual within the same region of birth as `n` takes the index number `n` and array `A` remains continuous.

```
Emigrate(int n)
{ int n2;
  N[A[n].state] -= 1;                    Decrement N[A[n].state].

  if(A[n].rob)                           Use emigrant's region of birth
  { n2 = ukbid-1; ukbid--; }             to find highest index number of
                                         individual who will take over
  else                                   emigrant's index number, to
  { n2 = immid-1; immid--; }             prevent array from having gaps
                                         of unoccupied index numbers.
  Transfer(n, n2);
}
```

1.15 Immigration generator

This routine brings a new immigrant into the population and schedules the next immigrant's arrival. The routine can be thought of as an external immigration event generator. The routine uses a pseudo-individual (index number `IMM`) to schedule the immigrant arrivals at evenly spaced intervals.

Entry: `t` contains the current time.
`t0` contains the end time of model.
`pimm` contains the proportion of immigrants who are non-UK born.
`immid` contains the next available index number for non-UK born.
`ukbid` contains the next available index number for UK born.
`IMM` contains the index number for the pseudo-individual used to schedule external immigration events handled here.
`ypi` contains the years per immigration, re-calculated each year from data on immigrants per year (`immig`).

Exit: An immigrant is brought into the population and the next immigrant due is scheduled.

```
ImmigrateG()
{ int y, n;
  y = (int)t - (int)t0;                          Get integer year array index.

  if(Rand()<pimm[y]) { n = immid; immid++; }     Determine whether immigrant will
  else               { n = ukbid; ukbid++; }     be UK or non-UK born.

  Immigrate(n);                                  Create immigrant.

  A[IMM].pending = pImmig;                       Schedule next immigration.
  EventSchedule(IMM, t+ypi);
}
```

1.16 Birth generator

This routine initiates a birth and schedules the next birth, at regularly spaced intervals, acting as the external event generator for births. The routine uses a pseudo-individual (index number `BIRTH`) to schedule births at evenly spaced intervals.

Entry: `t` contains the current time.
`ukbid` contains the next available ID number for UK-born.
`A[BIRTH]` is the individual designated for scheduling births.
`ypb` contains years per birth, re-calculated each year from data on births per year (`births`).

Exit: A new individual is born.
The next birth is scheduled.
`ukbid` is incremented.

```
BirthG()
{
  Birth(ukbid,t); ukbid++;          Produce a birth and increment the next
                                    available index number for UK-born.
  A[BIRTH].pending = pBirth;        Schedule the next birth for ypb
  EventSchedule(BIRTH,t+ypb);       years into the future.
}
```

1.17 Change states

This routine logs individuals out of states as they leave them and into new states as they enter. It maintains counters of the numbers in each state so that the array of individuals never has to be scanned for that information.

Entry: `n` indexes the individual.
`q` contains the new state. This is a number greater than zero, in the range `q0` to `q1`.
`A[n].state` contains the old state, either 0 or in the range `q0` to `q1`. If 0, this record is not in use.
`A[n].bstate` includes the non-susceptible states visited thus far.

Exit: `A[n].state` contains the number of the new state (`q` on entry).
`A[n].bstate` incorporates the new state (`q` on entry).
`A[n].tEntry` contains the time of entry to the new state.
`N[u]` is decremented, where `u` represents the old state.
`N[v]` is incremented, where `v` represents the new state.

```
NewState(int n, int q)
```

```
{
  if(q>qU)                                  Reduce the number in the old state
    N[A[n].state] -= 1;                     unless individual is entering Uninfected,
                                            which only happens at birth or immigration.
  if(N[A[n].state]<0)                       Make sure the state has not become
    Error2(609.0, "q=",(dec)q,              negative.
                  " n=",(dec)n);
  A[n].state  = q;                          Change state.
  N[A[n].state] += 1;                       Increase the number in the new state.
}
```

1.18 Transfer

This routine transfer all information about an individual, including saved event times, to a new identification number. The routine then cancels the pending event for that index number and re-schedules it using the new index number. The routine is used to keep the array of individuals contiguous for each region of birth.

Entry: `n` is the new index number to be assigned, which has no event scheduled
`n2` is the current index number of the individual.
There is an event scheduled for `n2`.

Exit: `n` is the new index number of the individual.
The event scheduled for n2 is now re-scheduled under `n` and all other data from `n2` are transferred to `n`. `n2` no longer has an event scheduled and the index number is free to be used again.

```
Transfer(int n, int n2)
{
  if(n!=n2)
  { A[n] = A[n2]; EventRenumber(n, n2); }     Copy data and reschedule as n.
}
```

1.19 Add reported case

This routine keeps track of the number of reported cases by age, sex, place of birth, disease site and calendar year. After a case is reported, they are scheduled for their next event. For the version of the model with a genetic component, this routine also allows for a stain to be genotyped and reports genotyping and other data for those cases, including the strain type identifier, and the age, sex, rob, etc for the case.

Entry: `t` is the current time.
`t0` is model start time.
`t1` is the model end time.
`A[n].tBirth` contains the time of birth.
`A[n].sex` contains the sex of the individual.
`A[n].rob` contains the region of birth of the individual, 0=non-UK, 1=UK born.
`SSAV` is the version of the model running, 0=non-SSA, 1=SSA.
`repc` holds the numbers of reported cases.
`repc2` holds the numbers of cases by place and time of infection.
`ptyped` holds the proportion of cases with genotyping data, by rob.
`repc3` holds data on cases reported and genotyped.
`A[n].tDeath contains the individuals` time of death.

A[n].tEmigrate contains the individual's time of emigration.
A[n].tExit contains the individual's time to remote infection.
A[n].tMutate contains the individual's strain mutation time.
A[n].ssa holds country of birth in SSA version of model,
0=UK and non-UK other, 1=SSA, 2=SSA,HIV+
A[n].tImm contains the time of immigration to UK.
A[n].tInfected contains the time infection was acquired.
A[n].inf contains the time and place of infection 1=recent/UK, 2=older/UK,
3=recent/non-UK, 4=older/non-UK.
A[n].strain contains the strain ID for the case.
A[n].state contains the disease state of the individual.

Exit: repc contains an additional case, individual n.
repc2 contains an additional case, if running REC version of the model.
repc3 contains an additional case, if n was selected for genotyping.
A[n].tRep contains a time past the end of the simulation, so that individual n will not be reported again.
Rep contains a status code.
1 A transmission is scheduled.
2 Recovery is scheduled.
3 Death is scheduled.
4 Strain mutation is scheduled.
5 Emigration is scheduled.

```
Rep(int n)
{
  int s,r,y,acl,d; dec age,wt, wd, we, wr, wm, wrep;
  age = t-A[n].tBirth;                          Get age.
  if(age<15) acl=0;                             Find age class (classes which match
  else if(age<45) acl=1;                        notification rates).
  else if(age<65) acl=2;
  else acl=3;
  s = A[n].sex;                                 Get sex.
  r = A[n].rob;                                 Get region of birth, 0=non-UK, 1=UK.
  y = (int)t - (int)t0;                         Get year for array index.
  if(A[n].state>=qD4) d=0;                      Get disease site (pulm/non-pulm)
  else d=1;                                     for arrray index.

  repc [acl][s][r][d][y] += 1;                  Increment reported cases.

  if(REC)                                       Increment reported cases for recent
  { repc2[acl][s][r][d][y][0]         += 1;     transmission stats, by inf.
    repc2[acl][s][r][d][y][A[n].inf] += 1; }

  if(Rand()<ptyped[r] && t>=2007)               If case is designated to be typed
  { repc3[repid][0] = age;                      and is within correct time window,
    repc3[repid][1] = s;                        store for genetic output.
    repc3[repid][2] = r;
    repc3[repid][3] = t;
    repc3[repid][4] = A[n].inf;
    repc3[repid][5] = A[n].strain;
    repid++;  }

  A[n].tRep = t1*2+Rand();                      Set reporting time distant enough
                                                that it cannot occur again.

  wd = A[n].tDeath;                             Get time of death.
  we = A[n].tEmigrate;                          Get time of emigration.
```

```
  wr = A[n].tExit;                          Get time to remote infection.
  wm = A[n].tMutate;                        Get strain mutation time.

  if(A[n].state<qD4)                        If this is pulmonary disease,
  { wt = A[n].tTransm;                      get time for transmission
    if(wt<wd && wt<we && wt<wr && wt<wm)     and if it occurs earliest,
    { A[n].pending = pTransm;               schedule it.
      EventSchedule(n, wt);
      return 1; }
  }

  if(wr<wd && wr<we && wr<wm)               If recovery occurs earliest,
  { A[n].pending = pRemote;                 schedule it.
    EventSchedule(n, wr);
    return 2; }

  if(wm<wd && wm<we)                        If mutation occurs earliest,
  { A[n].pending = pMutate;                 schedule it.
    EventSchedule(n, wm);
    return 4; }

  if(we<wd)                                 If emigration occurs earliest,
  { A[n].pending = pEmigrate;               schedule it.
    EventSchedule(n,we);
    return 5; }

  { A[n].pending = pDeath;                  Otherwise schedule death.
    EventSchedule(n, wd);
    return 3; }
}
```

1.20 Lifespan distribution

This routine assigns the number of remaining years to live for an individual, based on the present year, the individual's sex and age, and other factors. Various probability distributions may be selected. Exponentially distributed ages, with a constant chance of death in any year, are an option included for calibration of the model to results from an ordinary differential equation version of the model.

Entry: `sex` contains the individual's sex, 0=male, 1=female.
`age` contains the individual's present age, years and fractions thereof.
`mort` contains a mortality factor. For testing, this is the proportion of individuals who would die per year it deaths were strictly random (i.e., Poisson distributed in time).
`lifedsn` defines the lifespan distribution computation:
0 Exponential
1 Gompertz
2 Empirical life tables
`t` contains the present time.

Exit: `LifeDsn` contains the *remaining* life time (years until death) for the individual.

```
static int lifedsn = 2;                 Type of longevity distribution to be used.

dec LifeDsn(int sex, dec age, dec mort)
{ int yb, y, n; dec w;

  switch(lifedsn)
  {
```

```
    case 0: return Expon(mort);              Constant probability of death.
    case 2:                                  Empirical life tables.
    {  yb = (int)(t-age);                    Get year of birth.
       y  = yb-1870; if(y<0) y=0;            Get array index for birth year.
       w = RandF(A1, M1[y][sex], 122, age);
       return w; }
    default: Error(922.0); }                 Incorrect life span selection.

  return 0;                                  (Will never reach this.)
}
```

1.21 Emigration time distribution

This routine assigns the number of years until time of emigration from the study population for an individual, based on the present year, the individual's sex and age, and other factors in the condition of the individual. Exponentially distributed times, with a constant chance of emigration in each year, are included for calibration of the model to an ordinary differential equation version of the model.

Entry: `rob` contains the individual's region of birth, 0=non-UK, 1=UK
`sex` contains the individual's sex, 0=male, 1=female.
`age` contains the individual's present age, years.
`em` contains the emigration rate.
`emdsn` defines the emigration time distribution computation:
0 Exponential
2 Empirical migrant flow data.
`t` contains the present time.

Exit: `EmDsn` contains the *remaining* time in the UK for the individual in the updated version of the program. Note that many individuals will have dates of emigration beyond the running time of the model or beyond their own death date; these individuals will never emigrate.

```
static int emdsn = 0;                Type of longevity distribution to be used.

dec EmDsn(int rob, int sex, dec age, dec em)
{ int yb, y, a, n; dec w;

  switch(emdsn)
  {
    case 0: return Expon(em);                Constant probability of
                                             emigration.
    default: Error(922.0);                   Incorrect life span selection.
  }
  return 0;                                  (Will never reach this.)
}
```

1.22 Recovery distribution

This routine assigns a time to remote infection based on the present year, the individual's sex and age, and other factors in the condition of the individual. Various probability distributions may be selected.

Entry: `sex` contains the individual's sex, 0=male, 1=female.
`age` contains the individual's present age, years.

`r` contains a recovery parameter describing the individual. For testing this represents the proportion that would recover per year if recovery were strictly random (i.e., Poisson distributed in time).
`t` contains the present time.

Exit: `RecovDsn` contains the time until recovery, in years.

```
static dec recovdsn = 0;      Type of recovery distribution to be used.
static dec rmu      = 0.0;    Centering of recovery distribution, years.
static dec rsigma   = 0.1;    Half-width of recovery distribution, years.

dec RecovDsn(int s, dec age, dec r)
{ dec w;

  switch((int)recovdsn)                              Select the type of recovery.
  { case 0: return Expon(r);                         (completely random)
    case 1: w = 0;                          break;   (completely fixed)
    case 2: w = Uniform(-rsigma,rsigma);    break;   (uniform variation)
    case 3: w = LogNormal(rmu, rsigma);     break;   (log-normal variation)
    case 4: w = Gauss(.0, rsigma);          break;   (truncated Gaussian variation)
    case 5: w = Cauchy(.0, rsigma);         break;   (truncated Cauchy variation)
    default: Error(922.);
  }

  return max(1e-9, w+1./r);                          Return the time for recovery,
}                                                    always after a slight delay.
```

1.23 Time to disease

This routine assigns a time to disease based on the individual's age, sex, region of birth, infection status (Recent Infection, Remote Infection, Reinfection), and – in the SSAV version of the model – HIV status. Note, Recent Infection and Reinfection states are handled similarly, whereas Remote Infection is handled somewhat differently.

Notes on SSA version of model: Would like to get disease rates correct so things are comparable between the SSA and non-SSA models. I think it is best to leave them as a ratio (`ehiv`) and then when fitting model, the non-SSA model should fit higher disease risk for non-UK born than in the SSA model. In the SSA model they should be lower since a portion of non-UK born individuals will have elevated risk due to HIV.

Entry: `n` contains the individual's identifier.
`s` contains the individual's sex, 0=male, 1=female.
`a` contains the individual's integer age.
`rob` contains the individual's region of birth, 0=non-UK, 1=UK, 2=SSA
`tinf` contains the time since infection (years).
`d1` and `d3` contain the probability of progressing to disease for `qI1` and `qI3` over the first five years of infection.
`d2` contains the probability per year of progressing to disease for `qI2`.
`A[n].state` contains the present state of the individual, `qI1`, `qI2`, `qI3`.
`t` contains the present time.
`drr` gives the relative risk of disease over the first five years of infection.
`B1` contains the year of infection, for use with `drr`.
`SSAV` contains 1 if SSA version of model is to be run.

Exit: `Tdis` contains the number of years until disease development.

```
dec Tdis(int n, int a, int s, int rob, dec tinf)
{ dec d,age,w;
  if(SSAV && A[n].ssa==2)                  If running SSA version of model and
    rob=2;                                 individual is HIV+, adjust rob.

  switch(A[n].state)
  { case qI1:                              Process Recently Infection.
    { d = d1[s][rob][a]                    First calculate overall disease risk
                                           for tinf greater than 0 if applicable.

      if(Rand()>d)                         If disease should NOT occur, schedule
      { w = 2*RT+Rand();                   disease past the running time of
        return w; }                        model so that it never occurs.
      else
      { w = RandF(B1,drr,6,tinf);          If disease should occur, randomly
        return w; } }                      choose year, based on relative risk
                                           over five years.

    case qI3:                              Process Reinfection.
    { d = d3[s][rob][a]                    First calculate overall disease risk
                                           for tinf greater than 0 if applicable.

      if(Rand()>d)                         If disease should NOT occur, schedule
      { w = 2*RT+Rand();                   disease past the running time of
        return w; }                        model so that it never occurrs.
      else
      { w = RandF(B1,drr,6,tinf);          If disease should occur, get year
        return w; } }                      from relative risk over five years.

    case qI2:                              Process Remote Infection.
    { age = t-A[n].tBirth;
      w = RandF(A2,d2[s][rob],AC+2,age);
      return w;      }

    default: Error(922.);   }
  return 0;                                (Will never reach this.)
}
```

1.24 Get random age for immigrant

This routine assigns an age to an individual who is immigrating into the population. Age is randomly assigned based on probabilities of age classes from data. Probabilities of age classes are conditional on sex and region of birth.

Entry: `n` contains the individual's identifier.
`s` contains the individual's sex, 0=male 1=female.
`r` contains the individual's region of birth, 0=non-UK, 1=UK
`t` contains the present time.
`immage` contains cumulative probabilities of the age classes; to be compatible with future calls to `RandF`, the first cumulative probability is 0.
`SSAV` contains 1 if SSA version of model is to be run.

Exit: `GetAge` contains the age (in years) of the individual.

```
dec GetAge(int n, int s, int r)
{ dec rn,age; int y;

  rn = Rand();                             Get random number.
```

```
  if(SSAV)
    if(A[n].ssa) r=2;
```
If running SSA version of model, check if SSA and adjust `r` (rob) if so.

```
  y = (int)t - (int)t0;
```
Get array index for calendar year.

```
  if(rn<immage[y][s][r][1])
    return Rand()*15;
  if(rn<immage[y][s][r][2])
    return Rand()*10+15;
  if(rn<immage[y][s][r][3])
    return Rand()*10+25;
  if(rn<immage[y][s][r][4])
    return Rand()*10+35;
  if(rn<immage[y][s][r][5])
    return Rand()*15+45;
```
Assign a random age class within the correct age class, depending on the random number draw and cumulative probabilities of each age class specified by `immage`.

```
  age=Expon(0.10)+60;
  if(age>=121) age=120+Rand();
```
For the age class 60+, add age of 60 plus draw from exponential distribution with mean of 10 years.

```
  return age;
}
```

1.25 Parameter changing

This function updates variables associated with parameters that change with each model run. This routine must come after any change to parameters, e.g. through the `gparam` function. Currently four disease risk parameters are varied during model fitting: `d1uk20[M]`, `d2uk20[M]`, `d3uk20[M]` and `df`. `d1uk20[M]` is the risk of developing Primary Disease for UK-born males aged 20 years above. `d2uk20[M]` is the annual risk of developing Reactivation Disease for UK-born males aged 20 years and above. `d3uk20[M]` is the risk of developing Reinfection Disease for UK-born males aged 20 years and above. `df` is the factor by which UK-born disease risks are multiplied to get non-UK born risks.

Entry: `drr` contains relative cumulative rates of disease progression by time since infection, up to five years.
`d1uk20[M]` contains the risk of developing Primary Disease for UK-born males aged 20 years above
`d2uk20[M]` contains the annual risk of developing Reactivation Disease for UK-born males aged 20 years and above.
`d3uk20[M]` contains the risk of developing Reinfection Disease for UK-born males aged 20 years and above.
`df` contains the factor by which UK disease risk are multiplied to get non-UK born disease risks.
`sdf1` contains the sex ratios of female:male disease risk (Primary Disease).
`sdf2` contains the sex ratios of female:male disease risk (Reactivation Disease).
`sdf3` contains the sex ratios of female:male disease risk (Reinfection Disease).
`presp` contains the proportion of disease respiratory for children.

Exit: `d1`contains Primary Disease risk by sex, rob (UK/nonUK/HIV) and age.
`d2`contains Reactivation Disease rate by sex, rob (UK/nonUK/HIV) and age.
`d3`contains Primary Disease risk by sex, rob (UK/nonUK/HIV) and age.

NOTE. Disease progression risks are specified separately for Recent, Remote, and Reinfected individuals (`d1`, `d2`, `d3`), and also allowed to vary by sex, rob and age. Following Vynnycky and Fine, the age-dependency of risk is specified by two parameters – one for risk ages 0-10 (constant) and one for 20+ (constant). Risk between ages 10-20 is assumed

to increase linearly from the rate at 10 to the rate at age 20.For those over 10 and under 20 overall disease progression rate is: `A0+(age-10)*((A20-A10)/10)`.

For `d1` and `d3`, these represent overall, cumulative risks of disease progression in the first five years of infection or reinfection for infection at a given age. The array `drr` specifies the cumulative relative risk over these five years and is used along with `d1/d2` to generate a time to disease, if applicable. For `d2` the disease progression risks are annual rates at a given age (and sex/rob) for disease progression. `d2` is converted to cumulative risk by age, and the cumulative distribution is used, along with current age of individual infected, to assign a time to disease.

Disease progression risks estimated by Vynnycky and Fine 1997 for white ethnic males are used to set UK-born males and UK-born female risk for those under ten years of age. For all ages, female risks are calculated by multiplying male risks by the risk ratios for sex, `sdf1`, `sdf2`, and `sdf3`. Risks for 20 year-olds are allowed to vary in the model (one for each disease types, so three parameters). For non-UK born risks, `df` is multiplied by the UK-born risks to get non-UK born risks. `df` is allowed to vary in the model. HIV-positive risks are further multiplied by `ehiv`. This means as the three UK-born risks and `df` are varied during model fitting, non-UK born disease progression risks would need to be re-generated each time the model is run.

One complication regards pulmonary versus overall disease risk. In Vynnycky and Fine 1997, risks are for respiratory disease. However, disease risk needed for the model is overall disease risk, pulmonary plus non-pulmonary. For those fixed in the model (UK-born under 10), the respiratory risk is corrected to equal overall (pulmonary + non-pulmonary) risk.

```
Param()
{ int a,s,r;
  dec ep = 0.00000000000001;          Check that ehiv and df are not
  if(ehiv<ep) ehiv= ep;               negative, making them ep, a small
  if(df<ep)   df  = ep;               positive number if not.

  mi=.106*md;                         Set infection mutation rate after
                                      disease mutation rate is read.

  if(DPARAM)                          For new way of varying disease risks:
  { if(d1uk10[M]<ep) d1uk10[M]=ep;    First check that all risks/rates are
    if(d1uk20[M]<ep) d1uk20[M]=ep;    positive (greater than a small positive
    if(d2uk10[M]<ep) d2uk10[M]=ep;    number).
    if(d2uk20[M]<ep) d2uk20[M]=ep;
    if(d3uk10[M]<ep) d3uk10[M]=ep;
    if(d3uk20[M]<ep) d3uk20[M]=ep;

    d1uk10[F] = d1uk10[M]*sdf1[0];    Also for new version of model, set
    d2uk10[F] = d2uk10[M]*sdf2[0];    females' values for disease progression,
    d3uk10[F] = d3uk10[M]*sdf3[0];    relative to males, with sex ratios
    d1uk20[F] = d1uk20[M]*sdf1[1];    calculated from rates in Vynnycky
    d2uk20[F] = d2uk20[M]*sdf2[1];    Fine.
    d3uk20[F] = d3uk20[M]*sdf3[1];

    for(s=0; s<2; s++)
    { d1uk10[s] = d1uk10[s]/presp;    Also for new version of model, set
      d2uk10[s] = d2uk10[s]/presp;    divide by presp to get overall (not
      d3uk10[s] = d3uk10[s]/presp;    respiratory) progression rates/risks.
    }

    for(a=0; a<10; a++)               Now expand rates to all ages for
    for(s=0; s<2;  s++)               by assuming constant risk/rate from age
    { d1[s][UK][a] = d1uk10[s];       0-10, linear increase from 10-20, and
      d2[s][UK][a] = d2uk10[s];       constant risk/rate from age 20+. Note,
      d3[s][UK][a] = d3uk10[s]; }     for d1 and d3 these are cumulative risks
```

```
                                        over first 5 yrs for (infected at age a)
  for(a=10; a<20; a++)                  while for d2 these are annual rates of
  for(s=0;  s<2;  s++)                  progression.
  { d1[s][UK][a] = d1uk10[s] + (a-10)*((d1uk20[s]-d1uk10[s])/10);
    d2[s][UK][a] = d2uk10[s] + (a-10)*((d2uk20[s]-d2uk10[s])/10);
    d3[s][UK][a] = d3uk10[s] + (a-10)*((d3uk20[s]-d3uk10[s])/10); }

  for(a=20; a<121; a++)
  for(s=0;  s<2;   s++)
  { d1[s][UK][a] = d1uk20[s];
    d2[s][UK][a] = d2uk20[s];
    d3[s][UK][a] = d3uk20[s]; }
}

else                                    For old version of disease risk–
{ for(a=0; a<121; a++)                  For UK-born get overall disease risks
  for(s=0; s<2;   s++)                  (d1 and d3) and annual risks (d2) from
  { d1[s][UK][a] = d1p[a][s][UK]/presp; respiratory risks, taken from
    d2[s][UK][a] = d2p[a][s][UK]/presp; Vynn and Fine 1997 (divided by prop.
    d3[s][UK][a] = d3p[a][s][UK]/presp; }of all tuberculosis which is respiratory
}                                       (see Data). Indexed by sex, rob,
                                        and age.

for(a=0; a<121; a++)                    For non-UK born: multiply UK-born
for(s=0; s<2;   s++)                    risks by df to get non-UK born
{ d1[s][NUK][a] = df*d1[s][UK][a];      disease risks by sex, rob and age.
  d2[s][NUK][a] = df*d2[s][UK][a];
  d3[s][NUK][a] = df*d3[s][UK][a]; }

for(a=0; a<121; a++)                    Check that overall disease risks
for(s=0; s<2;   s++)                    (d1, d3) are not above 1; also check
{ if(d1[s][NUK][a]>1) d1[s][NUK][a] = 1; annual rates (d2) are not above 1
  if(d2[s][NUK][a]>1) d2[s][NUK][a] = 1; for non-UK born.
  if(d3[s][NUK][a]>1) d3[s][NUK][a] = 1; }

if(SSAV)
{ for(a=0; a<121; a++)                  Get disease risks for HIV-positive
  for(s=0; s<2;   s++)                  SSAs by multiplying non-UK born
  { d1[s][HIV][a] = ehiv*d1[s][NUK][a]; risks/rates by factor ehiv.
    d2[s][HIV][a] = ehiv*d2[s][NUK][a];
    d3[s][HIV][a] = ehiv*d3[s][NUK][a]; }

  for(a=0; a<121; a++)                  Make sure disease risks for HIV+
  for(s=0; s<2;   s++)                  are not above 1.
  { if(d1[s][HIV][a]>1) d1[s][HIV][a]=1;
    if(d2[s][HIV][a]>1) d2[s][HIV][a]=1;
    if(d3[s][HIV][a]>1) d3[s][HIV][a]=1; }
}

for(s=0; s<2; s++)                      Fix end of array for d2 (longer).
for(r=0; r<3; r++)
  d2[s][r][121] = d2[s][r][120];

for(s=0; s<2; s++)                      Redefine d2 as cumulative probability of
for(r=0; r<3; r++)                      disease progression; first translate risk
  d2[s][r][1] = d2[s][r][0];            in first year of life (d2[s][r][0]) as
                                        cumulative risk experienced before age 1
for(a=2; a<=121; a++)                   (d2[s][r][1]). Then convert rest of array
for(s=0; s<2; s++)                      to cumulative risks. Indexed by sex, rob
for(r=0; r<3; r++)                      and age.
```

```
    d2[s][r][a] = d2[s][r][a-1]+(1-d2[s][r][a-1])*d2[s][r][a];

  for(s=0; s<2; s++)                Make sure cumulative probability of disease
  for(r=0; r<3; r++)                has not gone beyond 1.
  { if(d2[s][r][121]>1) Error(754.1); }

  for(s=0; s<2; s++)                Finish cumulative distribution so that
  for(r=0; r<3; r++)                cumulative risk before age 0 is 0 and
  { d2[s][r][0] = 0.0;              those who should never progress to disease
    d2[s][r][122] = d2[s][r][121]; are assigned times far into the future so
    d2[s][r][123] = 1.0; }          that disease progression does not happen.

}
```

1.26 Initialize starting population

This function sets up the intial population by looping through matrix `n1981` which holds numbers in the initial population by age, sex, and rob. For the SSA version of model, `ssa1981` is used for the proportions of SubSaharan Africans among non-UK born in 1981 by age and sex.

Notes: In Param(), could multiply 1981 (if they are proportions) by initial pop size, e.g. `initpop`. Could make assignments deterministic since the numbers are so large – this would take some planning for dealing with remainders etc.

Entry: `n1981` contains numbers by age, sex, and rob for individuals in the population at initialization in 1981.
`ssa1981` contains the proportion of SSAs among non-UK born at population initialization.

Exit: The intial population is set up; each individual is assigned attributes and scheduled for exactly one event (other event times may be stored for an individual.

```
InitPop()
{ int a,s,i,n,st,rob; dec age,wd,we,wv,tinf;
  ukbid = maximm+1;                          Initialize ID numbers to
  immid = 1;                                 correct values.
```

NOTE: Could make function so that UK-born, non-UK born and SSA-born are initialized with one function, called three times. Also function so that states are assigned and processed in a function

```
  for(a=0; a<121; a++)                       First, initialize UK-born
  for(s=0; s<2;   s++)                       (rob=1) population for all age
  for(i=0; i<n1981[a][s][UK]; i++)           and sex categories.
  { n = ukbid; ukbid++;                      Take the next available ID.
    age = a+Rand();                          Assign age plus random bit.
    A[n].tBirth = t-age;                     Assign birth time from age.
    A[n].sex = s;                            Assign sex.
    A[n].rob = rob = UK;                     Set to UK-born.

    BasicInd(n,UK,age,s);                    Set up basic individual.
    DisState(n,UK,a); }                      Assign disease state and
                                             process accordingly.

  if(SSAV)                                   Process non-UK born in SSA
    for(a=0; a<121; a++)                     version of model, taking into
    for(s=0; s<2;   s++)                     account the proportion of
    for(i=0; i<n1981[a][s][NUK]; i++)        SSAs and their HIV
```

```
    { n = immid; immid++;                          status.
      age = a+Rand();
      A[n].tBirth = t-age;
      A[n].sex = s;
      A[n].rob = rob = NUK;
      if(Rand()<ssa1981[a][s])                     If SubSaharan African, indicate
      { A[n].ssa = 1;                              this and assign HIV status.
        rob=SSA;
        if(Rand()<hivp[s][0]) A[n].ssa=2; }

      BasicInd(n,NUK,age,s);                       Set up basic individual.
      DisState(n,rob,a); }                         Assign disease state and
                                                   process accordingly.

  else                                             Process non-UK born for non-SSA
    for(a=0; a<121; a++)                           version of model.
    for(s=0; s<2; s++)
    for(i=0; i<n1981[a][s][NUK]; i++)
    { n=immid; immid++;
      age=a+Rand();
      A[n].tBirth = t-age;
      A[n].sex = s;
      A[n].rob = rob = NUK;

      BasicInd(n,NUK,age,s);                       Set up basic individual.

      DisState(n,NUK,a); }                         Assign disease state and
                                                   process accordingly.
}
```

1.27 Set up basic individual for population initialization

This function sets up a basic individual assigned to a death, emigration, or vaccination time. This is similar to birth but processes individuals of any age, sex, or region of birth (anyone being initialized when the model starts). The scheduled event may change if a state other than `Uninfected` is assigned when diseaes state is assigned. This function is merely used to get the individual initialized.

Entry: `A[n]` individual is not scheduled for any event.
`rob` is 0 for non-UK born (ONUK and SSA), 1 for UK-born.
`age` is the age of the individual in years.
`sex` is 0 for males and 1 for females.

Exit: `A[n]` is in the Uninfected state and scheduled for its earliest event.

```
BasicInd(int n, int rob, dec age, int s)
{ dec wd, we, wv;

  NewState(n,qU);                                  Assign to Uninfected state.
  A[n].tDeath = wd = t+LifeDsn(s,age,m1[0][0]);    Assign time of death.
  if(wd<A[n].tBirth+age) Error(612.2);             Check death time.
  A[n].tEmigrate = we                              Assign time of emigration.
                 = t+EmDsn(rob,s,age,em[s][rob]);
  if(age<v3[rob] && Rand()<v1[rob]*v2[rob])        Calculate time to vaccination,
    wv = t+(v3[rob]-age)+Rand();                   set to time which never
  else wv = t+2*RT+Rand();                         happens if it should not occur.

  if(wv<wd && wv<we)                               If vaccination is the earliest
```

```
  { A[n].pending = pVaccin;                    event, schedule it.
    EventSchedule(n, wv); }

  else if(wd<we)                               If death is the earliest
  { A[n].tExit = wd;                           event, schedule it.
    A[n].pending = pDeath;
    EventSchedule(n, wd); }
  else                                         Or, if emigration is the
  { A[n].tExit = we;                           earliest event, schedule that.
    A[n].pending = pEmigrate;
    EventSchedule(n, we); }
}
```

1.28 Assign disease state for initial member of population

This function assigns one of the eight disease states (8 instead of 11 because those with disease are not assigned to pulmonary or non-pulmonary until after being processed).

Entry: `n` is the individual's ID number.
`A[n]` is set up as basic `Uninfected` individual.
`rob` is the region of birth, 0=Non-UK, 1=UK, 2=SSA (if running `SSAV` version of model.
`a` is the integer age of the individual.
`Ax` contains the disease state, used for `RandF` along with `inf1981`.
`inf1981` contains the probabilities of the different disease states, which are numbers in `Ax`.

Exit: `A[n]` has been assigned disease state (may remain unchanged) and processed accordingly.

```
DisState(int n, int rob, int a)
{ int st,str,r; dec tinf;
  st = 1+(int)RandF(Ax,inf1981[a][A[n].sex][rob],9,1.0); Get disease state.

  r=rob;                                       Get rob of 0/1 for assigning
  if(r==2) r=0;                                place of infection in inf
                                               and for assigning strain ID.

  switch(st)
  {
    case 2:                                    Send to vaccination.
      EventCancel(n);
      Vaccination(n);
      break;

    case 3:                                    Get the time of infection before
      tinf = Rand()*5;                         sending to Infect.
      Infect(n,tinf,Strain(r),0);
      if(REC)                                  Assign inf (time/place)
      { if(r) A[n].inf = 1;                    if REC version of model.
        else  A[n].inf = Rand()<.5? 1: 3; }
      break;

    case 4:                                    Before sending to Remote, put
      EventCancel(n);                          in temporary disease state to
      A[n].strain = Strain(r);                 facilitate re-scheduling of
      NewState(n,qD1);                         events in Remote function.
      Remote(n);
      if(REC)                                  Assign inf (time/place)
```

```
      { if(r) A[n].inf = 2;                    if REC version of model.
        else  A[n].inf = Rand()<.25? 2:4; }
      break;

    case 5:                                    Before sending to Infect, put
      NewState(n,qI2);                         in "Remote infection" state so
      tinf = Rand()*5;                         that this is correctly treated
      Infect(n,tinf,Strain(r),0);              as Reinfection.
      if(REC)                                  Assign inf (time/place)
      { if(r) A[n].inf = 1;                    if REC version of model.
        else  A[n].inf = Rand()<.5? 1: 3; }
      break;

    case 6: case 7: case 8:                    Process disease states by first
      EventCancel(n);                          putting in correct infection
      A[n].strain=Strain(r);                   state and then sending to
      NewState(n,st-3);                        Disease().
      Disease(n);
      if(REC)                                  Assign inf (time/place)
      { if(r&&st==7) A[n].inf = 2;             if REC version of model.
        else if(r)   A[n].inf = 1;
        else if(st==7) A[n].inf = Rand()<.2? 2: 4;
        else           A[n].inf = Rand()<.4? 1: 3; }
      break;

    default:
      if(st!=1) Error(618.2);
  }
}
```

1.29 Choose strain identification number

Future versions of the model will track strain type profile identifiers for all infections. This is a stub for a future routine to choose the strain number at model initialization or for immigrants at immigration. It is a placeholder and entry/exit conditions have not been determined yet.

Entry: `type` holds the type of strain number to be generated, where 0=Non-UK born at model initialization and migrants during the simulation, 1=UK-born at model initialization.
`S0` holds strain IDs for migrants, corresponding to probabilities held in `sdimm`.
`S1` holds strain IDs for UK-born at model initialization, which correspond to probabilities held in `sduk`.
`sdimm` holds cumulative probabilities of selection for strains in migrants, `S0`.
`sduk` holds cumulative probabilities of selection for strains in UK-born at model initialization, held in `S1`.
`is0` holds the number of non-UK born strains, for model initialization and for migrants upon entry to the UK.
`is1` holds the number of UK-born strains at model initialization.

Exit: A strain ID number is returned.

```
int Strain(int type)
{
  if(type==0)                                  Process Non-UK at model initialization,
```

```
    return (int) RandF(S0,sdimm,is0+1,1);     where strains are numbered 1 to is0.
  else                                        Process UK-born at model initialization,
    return (int) RandF(S1,sduk,is1+1,is0+1);  where strains are numbered is0+1 to is0+is1.
  return 0;
}
```

1.30 Cluster analysis

This function analyzes genetic typing data created during the simulation, checking whether cases have strains that match others in the population and computing cluster sizes for each strain. This routine is called once at the end of the simulation, so speed is not an issue.

Entry: `repc3` holds data on cases which are reported and genotyped; the first array dimension holds reporting identification numbers; the second dimension has 7 elements:
- 0 age
- 1 sex
- 2 birthplace
- 3 time of case report
- 4 place/time of infection (as in `A[n].inf`)
  - 1 recent/UK
  - 2 older/UK
  - 3 recent/NonUK
  - 4 older/NonUK
- 5 strain type of infection
- 6 set to zero.

`repid` holds the total number of reported cases.
`clust` is empty and set to zero, indexed as described its data definition.

Exit: `repc3` is updated to include the number of individuals with the same strain in element 6 of the second array dimension.
`clust[ac][s][r][k]` counts the number of cases for each combination of age class (0=0 14, 1=15 44, 2=45 64, 3=65), sex (0=male, 1=female), and rob (0=foreign, 1=UK) combination, with the fourth array dimension counting where and when infected:
- 0 unique and non-recent/non-UK,
- 1 unique and recent/UK cases,
- 2 clustered and non-recent/non-UK,
- 3 clustered and recent/UK,
- 4 total of indexes 0 to 3.

```
int Clust()
{ int i,j,k,ac,s,r; dec age;
  for(i=0; i<repid; i++)                      For each case count the
  { for(j=0; j<repid; j++)                    number of other cases
      if (j!=i && repc3[i][5]==repc3[j][5])   with matching strains.
        repc3[i][6]++;
    age = repc3[i][0];                        Get age class, sex, and
    ac  = age<15?0: age<45?1: age<65?2: 3;    birthplace indexes.
    s   = repc3[i][1];
    r   = repc3[i][2];
    k = repc3[i][4]==1? 1: 0;                 Develop the array index for the
```

```
    if(repc3[i][6]>0) k += 2;
```
fourth dimension, by recent/UK and clustering.

```
    clust[ac][s][r][k]++;
    clust[ac][s][r][4]++; }
```
Increment cases by category and accumulate the total.

```
  return 0;
}
```
Return to caller.

1.31 Reporting

This function is called periodically to display the cumulative number of infections and other statistics. This function should be called no less than once per year, in current set-up, so that births per year (`bpy`) and immigrants per year (`ipy`) can be updated at least each year. For generating mid-year population estimates, this function should be called at least twice per year, once at the beginning of the year and once mid-way through the year.

Entry: `prog` contains the name of the program presently running.
`t` contains the current time.
`t0` contains the starting time.
`N` contains the value of each dynamical variable, index by the state number (`qU`, `qV`, `qI1`, and so forth).
`startsec` contains the wall-clock time of the start of the run.
`kernel` defines the contagion kernel.
`deaths` and `events` contain the number of deaths and events since the counters were cleared.
`rand0` defines the random number sequence used.

Exit: The next line of the report has been printed.
`deaths` and `events` are cleared.

```
static int ReportFirst;
ReportInit() { ReportFirst = 0; }

Report(char *prog)
{
  int i,ac,s,r,y,yr; dec z,age;
  if(ReportFirst==0)
  { ReportFirst = 1;
    printf("Dataset:     Simulation output of program '%s'\n", prog);
    printf("Kernel:      %s\n",
      kernel==0? "Mean field":
      kernel==1? "Cauchy":
      kernel==2? "Gaussian":
                 "Unspecified");
    printf("Sequence:    %lu\n\n", rand0);

    EventProfile("Initial");

   printf("Label t:       Time, in years and fractions thereof.\n");
   printf("Label N:       Total population size.\n");
   printf("Label Up:      Prevalence of susceptible individuals.\n");
   printf("Label Vp:      Prevalence of immune individuals.\n");
   printf("Label I1p:     Prevalence of new infections.\n");
   printf("Label I2p:     Prevalence of latent infections.\n");
   printf("Label I3p:     Prevalence of reinfections.\n");
   printf("Label D1p:     Prevalence of new/primary disease.\n");
   printf("Label D2p:     Prevalence of reactivation disease.\n");
   printf("Label D3p:     Prevalence of reinfection disease.\n");
   printf("Label D4p:     Prevalence of primary non-pulmonary disease.\n");
   printf("Label D5p:     Prevalence of react non-pulmonary disease.\n");
```

```
 printf("Label D6p:     Prevalence of reinf non-pulmonary disease.\n");
 printf("Label U:       Number of susceptible individuals.\n");
 printf("Label V:       Number of immune individuals.\n");
 printf("Label I1:      Number of new infections.\n");
 printf("Label I2:      Number of latent infections.\n");
 printf("Label I3:      Number of reinfections.\n");
 printf("Label D1:      Number of new/primary disease.\n");
 printf("Label D2:      Number of reactivation disease.\n");
 printf("Label D3:      Number of reinfection disease.\n");
 printf("Label D4:      Number of primary non-pulmonary disease.\n");
 printf("Label D5:      Number of react non-pulmonary disease.\n");
 printf("Label D6:      Number of reinf non-pulmonary disease.\n");

 printf("Label Deaths:  Number of deaths since last report.\n");
 printf("Label Events:  Number of events dispatched since last report.\n");
 printf("Label Elapsed: Seconds of elapsed wall-clock time to this point.\n");

  printf("\n|t      |N        |Up       |Vp       "
                  "|I1       |I2       |I3       "
                  "|D1       |D2       |D3       "
                  "|D4       |D5       |D6       "
                  "|U        |V        "
                  "|I1       |I2       |I3       "
                  "|D1       |D2       |D3       "
                  "|D4       |D5       |D6       "
                  "|Deaths   |Events   |Elapsed \n");
}

for(z=0,i=q0; i<=q1; i++) z += N[i];        Get population size.

printf("|%6.1f|%8.0f|%f|%f|%f|%f|%f|%f|%f|%f|%f|%f|%f"
       "|%8.0f|%8.0f|%8.0f|%8.0f|%8.0f|%8.0f|%8.0f|%8.0f"
       "|%8.0f|%8.0f|%8.0f|%8d|%8d|%5d\n",
  t, z,
  N[qU]/z, N[qV]/z, N[qI1]/z, N[qI2]/z, N[qI3]/z,
                    N[qD1]/z, N[qD2]/z, N[qD3]/z,
                    N[qD4]/z, N[qD5]/z, N[qD6]/z,
                    N[qU],    N[qV],
                    N[qI1],   N[qI2],   N[qI3],
                    N[qD1],   N[qD2],   N[qD3],
                    N[qD4],   N[qD5],   N[qD6],
                    deaths, events, (int)(time(NULL)-startsec));

fprintf(stderr, "  %.1f\r", t);            Update status indicator.
fflush(stdout); fflush(stderr);            Make sure everything shows.
deaths = events = 0;                       Clear time-step counters.

y = (int)t;                                Get calendar (integer) year.

if(y>lup)                                  Check to see if parameters
{ ypb = 1./bcy[y-(int)t0];                 sensitive to calendar year
  ypi = 1./immig[y-(int)t0];               need updating.
  lup = y; }

if((t-y)>0.3 && (t-y)<0.7 && y>2006)       Since computation is expensive
{                                          only get population sizes when
                                           necessary (mid-year).
  yr = y-(int)t0;                          Get year array index.

  for(i=1; i<immid; i++)                   Loop through immigrants.
  { r=0;                                   If running SSA version of
    if(SSAV  A[i].ssa) r=2; //model, find out if SSA.
    age = t-A[i].tBirth;                   Get age, sex, find age class.
    ac = age<15?0: age<45?1: age<65?2: 3;
    s=A[i].sex;
```

```
      N2[ac][s][r][yr] +=1;                    Increment correct compartment
      if(A[i].state>2)                         for this individual for
        Np[ac][s][r][yr] +=1;    }             population sizes and infection
                                               prevalence.

    r=1;                                       Loop through UK-born.
    for(i=maximm+1; i<ukbid; i++)
    { age = t-A[i].tBirth;                     Get age, sex and find age class.
      ac = age<15?0: age<45?1: age<65?2: 3;
      s = A[i].sex;
      N2[ac][s][r][yr] += 1;                   Increment correct compartment
      if(A[i].state>2)                         for this individual for
        Np[ac][s][r][yr] +=1;    }             population sizes and infection
                                    }          prevalence.
}
```

NOTES: Population sizes by age, sex, region of birth, year: Best way is probably to schedule in advance an `ageing event` which would depend on their age and the relevant age class divisions so that only the next one is scheduled of course. This may also have implications for STD modelling, for example. Alternatively, this could be done by having an array of individuals stored in order of birth year, as Clarence and I worked out in regard to age-dependent mixing. This array would be accompanied by information on where the first and last member of each age class (e.g. year of birth) is. One could quickly get the population sizes anytime during model run that way. However, here it will be done by simply looping through the entire array, `A[n]`, of individuals whenever population sizes are needed. This way things are ready in a short time, and it can be modified in the future.

1.32 Closure

When processing is finished, this routine writes final statistics, prints time trajectory of disease cases and returns an array with case notification rates (or numbers of notifications) observed over the simulation.

Entry: `startsec` contains the starting time, in the format of function `time`.
`trho` and `nrho` contain the total dispersal distance and the number of dispersal events.
`tinfections` and `linfections` contain the total number of infections targetted and the number that fell within the geographic area.
`tstep` has accumulated statistics throughout the run.

Exit: Final statistics have been displayed.
`out` contains the notification rates observed over the simulation.

```
static dec nstep  =  0;                    Number of steps
static dec tstep1 =  0;                    Mean time step
static dec tstep2 =  0;                    Standard deviation in time step
static dec tsmin  =  1E10;                 Smallest time step
static dec tsmax  = -1E10;                 Largest time step

static dec trho, nrho;                     Statistics for local dispersal.
static dec tinfections, linfections;

FinalInit()
{
  nstep = tstep1 = tstep2 = 0;
  tsmin  =  1E10; tsmax  = -1E10;
```

```
  trho = nrho = 0;
  tinfections = linfections = 0;
}

Final()
{ int a,s,r,y,d,den; dec size,w,w0,w1,w2,w3,w4,w5,w6,w7,w8,w9;
  FILE *pop;                                    Test output file.
  dec tot,tot2;                                 Population and case totals
                                                for printing aggregated rates.
  dec ratio;                                    Ratio of population sizes
                                                observed to those produced by the
                                                simulation.
  int inf;                                      Integer for repc2 index.
  printf("\n");
  size  = (indiv+3) * sizeof(struct Indiv);
  size += EventProfile("Final");

  tstepfin();
  { printf("Time steps:      Mean %s, Min %s, Max %s, SD %s, N %.0f\n",
           Tval(tstep1), Tval(tsmin), Tval(tsmax), Tval(tstep2), nstep); }

  if(nrho)
    printf("Dispersal:       Mean distance %.1f grid units.\n", trho/nrho);

  { printf("Infections:      Targeted %.0f, out of area %.0f, ratio %.2f%%\n",
      tinfections, tinfections-linfections,
      100.*(tinfections-linfections)/tinfections); }

  if(agec[0])
  { age1[0] /= agec[0]; age2[0] = sqrt(age2[0]/agec[0] - age1[0]*age1[0]);
    printf("All individuals: Mean age %.1f, SD %.1f, N %.0f\n",
      age1[0], age2[0], agec[0]); }

  if(agec[1])
  { age1[1] /= agec[1]; age2[1] = sqrt(age2[1]/agec[1] - age1[1]*age1[1]);
    printf("Disease-free:    Mean age %.1f, SD %.1f, N %.0f\n",
      age1[1], age2[1], agec[1]); }

  printf("\n");
  printf("Memory usage:    %.2f gigabytes\n", size/(1024*1024*1024));

  printf("Elapsed time:    %s\n",
    Tval((dec)(time(NULL)-startsec)/60/60/24/365.25));

  fprintf(stderr, "          \n");
  fflush(stdout); fflush(stderr);
  ifndef main
  include "plotting.c"
  endif

  printf("stid=%d\nrepid=%d\n",stid,repid); fflush(stdout);  Print stid
                                                and repid to find out how
                                                many strain mutations and reported
                                                cases occured.
```

Print notification rates by rob, year, sex, age

```
 outi=0;
 printf("Printing unadjusted notification rates by age, sex, and rob\n");
 printf("M,0-14\tM,15-44\tM,45-64\tM,65+\tF,0-14\tF,15-44\tF,45-64\tF,65+\nr=0\n");
 for(r=0; r<=1; r++)
```

```
{ printf("r=%d\n=notif\t",r);
  for(y=(tdata-(int)t0); y<RT; y++)
   { for(s=0; s<2; s++)
     for(a=0; a<4; a++)
     { w=100000*(repc[a][s][r][0][y]+repc[a][s][r][1][y])/N2[a][s][r][y];
       printf("%f\t ",w);
       out[outi++]=w; }
     printf("\n=notif\t"); }
  printf("\n\nr=%d\n=notif\t",r); }
printf("\n");
```

Calculate overall notification rates for each rob

```
outoi=0;
for(r=0; r<=1; r++)
{ w1=0; w2=0; w3=0;
  for(y=(tdata-(int)t0); y<RT; y++)
  for(s=0; s<2; s++)
  for(a=0; a<4; a++)
  for(d=0; d<2; d++)
  {  w1+= repc[a][s][r][d][y];
     if(d==0) w2+= N2[a][s][r][y]; }
  outo[outoi++]= w3=100000*w1/(w2>0? w2: 1);
  printf("=overall r=%d cases=%f pop=%f rate=%f\n",r,w1,w2,w3); }
```

Calculate one overall notification rate

```
w1=0; w2=0; w3=0;
for(r=0; r<=1; r++)
for(y=(tdata-(int)t0); y<RT; y++)
for(s=0; s<2; s++)
for(a=0; a<4; a++)
for(d=0; d<2; d++)
{  w1+=repc[a][s][r][d][y];
   if(d==0) w2+= N2[a][s][r][y]; }
outo[outoi++]=w3=100000*w1/(w2>0? w2: 1);
printf("=overall cases=%f pop=%f rate=%f\n",w1,w2,w3);
```

1.33 Print clustering proportions and other clustering output

For reference, clust's fourth array dimension is coded as follows: 0 unique and non-recent/non-UK, 1 unique and recent/UK cases, 2 clustered and non-recent/non-UK, 3 clustered and recent/UK, 4 total of indexes 0 to 3.

NOTE: The method here is for congruence with the array structure for notifications. For simplicity, some calculations are left redundant, but this has no significant effect on speed.

```
outci=0;
printf("Prop clustered and prop recent (of clustered) by age/sex/rob:\n");
printf("M,0-14\tM,15-44\tM,45-64\tM,65+\tF,0-14\tF,15-44\tF,45-64\tF,65+\n");

for(r=0; r<=1; r++)
{ printf("\nr=%d\n=clust\n",r);
  for(y=0; y<4; y++)
  { for(s=0; s<2; s++)
    for(a=0; a<4; a++)
    { w0=clust[a][s][r][2]+clust[a][s][r][3]; Total clustered.
```

```
      w1=clust[a][s][r][0]+clust[a][s][r][1]; Total unique.
      w2=clust[a][s][r][1]+clust[a][s][r][3]; Total recent/UK.
      w3=clust[a][s][r][0]+clust[a][s][r][2]; Total non-recent/non-UK.

      w8=clust[a][s][r][4]; w8=w8>0? w8: 1;   Total cases typed.
      w4=w0/w8;                               Proportion clustered.
      w5=w2/w8;                               Proportion recent/UK.
      w6=(dec)clust[a][s][r][3]/(w0>0? w0:1); Proportion of clustered cases
                                              which are also recent/UK.
      w7=(dec)clust[a][s][r][0]/(w1>0? w1:1); Proportion of unique which
                                              are also non-recent or non-UK.

      outc[outci++] = y==0? w4:               Fill one-dimensional output
                      y==1? w5:               array with the results and
                      y==2? w6:               record them on stdout.
                            w7;
      if(y==0)
        printf("a=%d,s=%d,r=%d\t %f\t %f\t %f\t %f\t\n",
                a,   s,   r,      w4, w5,  w6,  w7); } }
}
printf("\n");                                 Leave a blank line between groups.
```

Calculate overall proportion clustered for each rob

```
for(r=0; r<=1; r++)
{ w1=0; w2=0; w3=0;
  for(s=0; s<2; s++)
  for(a=0; a<4; a++)
  { w1+=clust[a][s][r][2];              Total clustered.
    w1+=clust[a][s][r][3];
    w2+=clust[a][s][r][4]; }            Total cases typed.
  outo[outoi++]=w3=w1/(w2>0? w2: 1);
  printf("=overall CLUST r=%d rate=%f\n",r,w3); }
```

Calculate one overall proportion clustered

```
w1=0; w2=0; w3=0;
for(r=0; r<=1; r++)
for(s=0; s<2; s++)
for(a=0; a<4; a++)
{ w1+=clust[a][s][r][2];     Total clustered.
  w1+=clust[a][s][r][3];
  w2+=clust[a][s][r][4]; }   Total cases typed.
outo[outoi++]=w3=w1/(w2>0? w2: 1);
printf("=overall CLUST =%f\n",w3);

printf("=overall outoi=%d\n\n",outoi);
```

Calculate proportion due to recent transmission (of all rep cases) for each rob

```
for(r=0; r<=1; r++)
{ w0=0; w1=0;
  for(y=(tdata-(int)t0); y<RT; y++)
  for(s=0; s<2; s++)
  for(a=0; a<4; a++)
  { w0+=repc2[a][s][r][0][y][0];        Total cases.
    w0+=repc2[a][s][r][1][y][0];
    w1+=repc2[a][s][r][0][y][1];        Total recent/UK cases.
    w1+=repc2[a][s][r][1][y][1]; }
```

```
  outo[outoi++]=w1/(w0>0? w0:1);  }      Proportion recent (method one).
```

Calculate overall proportion due to recent transmission (of all rep cases) one over

```
w0=0; w1=0;
for(r=0; r<=1; r++)
for(y=(tdata-(int)t0); y<RT; y++)
for(s=0; s<2; s++)
for(a=0; a<4; a++)
{ w0+=repc2[a][s][r][0][y][0];          Total cases.
  w0+=repc2[a][s][r][1][y][0];
  w1+=repc2[a][s][r][0][y][1];          Total recent/UK cases.
  w1+=repc2[a][s][r][1][y][1]; }
outo[outoi++]=w1/(w0>0? w0:1);          Proportion recent (method one).
```

Calculate proportion of typed cases which are due to recent transmission (by rob)

```
for(r=0; r<=1; r++)
{ w2=0; w8=0;
  for(s=0; s<2; s++)
  for(a=0; a<4; a++)
  { w2+=(clust[a][s][r][1]+clust[a][s][r][3]); Total recent/UK.
    w8+=clust[a][s][r][4]; }                   Total cases typed.
  outo[outoi++]=w2/(w8>0? w8:1); }             Proportion recent/UK.
```

Calculate proportion of typed cases which are due to recent transmission (overall)

```
w2=0; w8=0;
for(r=0; r<=1; r++)
for(s=0; s<2; s++)
for(a=0; a<4; a++)
{ w2+=(clust[a][s][r][1]+clust[a][s][r][3]); Total recent/UK.
  w8+=clust[a][s][r][4]; }                   Total cases typed.
outo[outoi++]=w2/(w8>0? w8:1);               Proportion recent/UK.
```

Print 'repc2' for information on recent transmission

```
if(REC)
{ printf("Printing rec (repc2) and derivatives by age, sex, rob and year:\n");
  printf("M,0-14\tM,15-44\tM,45-64\tM,65+\tF,0-14\tF,15-44\tF,45-64\tF,65+\n");
  printf("TotalCases\tTotalRecent\tTotalOlder/UK\tPropRecent\tPropOlder/nonUK\n");
  for(r=0; r<2; r++)
  for(y=(tdata-(int)t0); y<RT; y++)
  { printf(" Year=%d:",y);
    for(s=0; s<2; s++)
    for(a=0; a<4; a++)
    { w0 = repc2[a][s][r][0][y][0]+      Total cases.
           repc2[a][s][r][1][y][0];
      w1 = repc2[a][s][r][0][y][1]+      Total recent/UK cases.
           repc2[a][s][r][1][y][1];
      w2 = repc2[a][s][r][0][y][2]+      Total older/UK cases.
           repc2[a][s][r][1][y][2];
      w3 = repc2[a][s][r][0][y][3]+      Total recent/non-UK cases.
           repc2[a][s][r][1][y][3];
      w4 = repc2[a][s][r][0][y][4]+      Total older/non-UK cases.
           repc2[a][s][r][1][y][4];
      w5 = w2+w3+w4;                     Total older or non-UK cases.
      w6 = w1/(w0>0? w0:1);              Proportion recent (method one).
```

```
        w7 = w1/(w1+w2+w3+w4);              Proportion recent (method two).
        w8 = w5/(w0>0? w0:1);               Proportion cases older/nonUK (one).
        w9 = w5/(w1+w2+w3+w4);              Proportion older/nonUK (two).
        printf("\nr=%d\n=rec\t%.1f\t%.1f\t%.1f\t\t%.1f\t%.1f\n=rec",r,w0,w1,w2,w6,w8
    }
    printf("\n=rec"); }
```

Adjustment of `repc` (reported cases in model) and `repc2` (cases according to whether they are due to recent transmission or not) based on population sizes observed in England and Wales versus population sizes produced by the simulation.

```
  printf("Now printing N3 for testing \n");
  FileIO("N3test.txt", fmt[21], "w|");
  FileIO("repc1.txt", fmt[22], "w|");          Print reported cases before
                                               adjustment.
  for(y=tdata-(int)t0; y<RT; y++)              Loop over reported cases
  for(r=0; r<2; r++)                           by year,region of birth, sex,
  for(s=0; s<2; s++)                           and age class to correct for
  for(a=0; a<4; a++)                           observed versus actual
  { if(N2[a][s][r][y]==0)                      population sizes, taking care
      N2[a][s][r][y]=1;                        to avoid dividing by zero.
    ratio = N3[a][s][r][y]/N2[a][s][r][y];
 pop = fopen("pop.txt","w");                   Population sizes for checking.
 if(pop==0) printf("error opening test file\n"); fflush(stdout);

 fprintf(pop,"Printing population sizes by rob, age, sex, and year for N2 (model) an
 fprintf(pop,"N2(model pop sizes)\n");

 for(y=0; y<RT; y++)                           Print population sizes
 { for(r=0; r<2; r++)                          by year,region of birth, sex,
   { fprintf(pop,"%d:r=%d\t",y+(int)t0,r);     and age class to pop file.
     for(s=0; s<2; s++)
     for(a=0; a<4; a++)
       fprintf(pop, "%f\t",N2[a][s][r][y]);
     fprintf(pop, "\n"); }
   fprintf(pop, "\n"); }

 fprintf(pop,"N3(observed pop sizes)\n");
 for(y=0; y<RT; y++)                           Print population sizes
 { for(r=0; r<2; r++)                          by year,region of birth, sex,
   { fprintf(pop,"%d:r=%d\t",y+(int)t0,r);     and age class to pop file.
     for(s=0; s<2; s++)
     for(a=0; a<4; a++)
       fprintf(pop, "%f\t",N3[a][s][r][y]);
     fprintf(pop, "\n"); }
   fprintf(pop, "\n"); }
 printf("Printing NUMBERS of notifications\n");
 for(r=1; r>=0; r--)                           Print numbers of notifications
 { printf("For rob=%d: M,0-14\tM,15-44\tM,45-64\tM,65+\tF,0-14\tF,15-44\tF,45-64\tF,
   for(y=(tdata-(int)t0); y<RT; y++)           by year, rob, and sex, to be
   { printf("-");                              captured with grep and -.
     for(a=0; a<4; a++)
     for(s=0; s<2; s++)
       printf("\t%f",repc[a][s][r][0][y]+repc[a][s][r][1][y]);
     printf("\n"); }
   printf("\n");
 }
 printf("\n");
```

```
}
```

1.34 Timing statistics

The system can take very small time steps or very large, depending on the number and frequency of events. This routine keeps track of individual time steps for statistical tracking, to be reported at the end of the run.

Entry: `t` contains the current time.
`tn` contains the next instant of time.

Exit: Statistics have been accumulated for the time step.

```
tstep(dec t, dec tn)
{ dec dt;
  dt = tn-t;                          Compute the length of the step.
  tstep1 += dt;                       Accumulate the sum and sum of
  tstep2 += dt*dt;                    squares.
  if(tsmin>dt) tsmin = dt;            Accumulate the minimum and
  if(tsmax<dt) tsmax = dt;            maximum.
  nstep  += 1;                        Accumulate the count.
}
```

TIMING RESULTS

Entry: `tstep` has accumulated statistics throughout the run.

Exit: `nstep` contains the number of time steps.
`tstep1` contains the length of the average time step, in years.
`tstep2` contains the root-variance in length of the time steps.
`tsmin` contains the shortest time step.
`tsmax` contains the longest time step.

NOTE: Here it is called "root variance" rather than "standard deviation" because the division is by `n`, not `n-1`.

```
tstepfin()
{
  if(nstep==0) return;                Avoid division by zero.
  tstep1 /= nstep;                    Compute the mean.
  tstep2 = tstep2/nstep - tstep1*tstep1;   Compute the variance.
  tstep2 = sqrt(tstep2);              Convert to root-variance.
}
```

1.35 Check for monotonicity

This routine checks whether a table of cumulative probabilities is monotonically increasing and optionally whether it is bracketed by 0 and 1.

Entry: `p` is the table of cumulative probabilities.
`n` is the number of entries in the table.
`b` is set if the table should begin with 0 and end with 1.
`r1` and `r2` contain two numbers that may help to identify the location of the error. If such numbers will not help, then either or both contain zero.

Exit: The routine returns if the table appears to be correct. If not, an error message is issued and the routine never returns.

```
int monotone(dec p[], int n, int b, int r1, int r2)
{ int i;
```

```
  for(i=1; i<n; i++)                          Make sure the sequence never
    if(p[i-1]>p[i])                           decreases.
      Error3(621., "‘",r1, "‘ ",r2, "‘ ",i);
```

```
  if(b && (p[0]!=0||p[n-1]!=1))               If requested, make sure it begins
    Error2(622., "‘",r1, "‘ ",r2);            with 0 and ends with 1.
```

```
  return 0;                                   (Will never reach this).
}
```

1.36 Service routines

```
char *pntab[] =                               Table of parameter names.
{ "s2[0]","s2[1]","c[0][0]","c[0][1]","c[1][0]", "c[1][1]",
  "v1[0]","v1[1]", "v2[0]", "v2[1]",  "v3[0]",   "v3[1]","ehiv",
  "r1[0]","r1[1]", "r2[0]", "r2[1]",  "r3[0]",   "r3[1]",
  "r4[0]","r4[1]", "r5[0]", "r5[1]",  "r6[0]",   "r6[1]",
  "r7[0]","r7[1]", "r8[0]", "r8[1]",   "df",     "runid",
  "d1uk20","d2uk20","d3uk20", "md",    "mi",
  "pmale[0]", "randseq", 0 };
```

```
dec *patab[] =                                Table of parameter addresses.
{ &s2[0], &s2[1],  &c[0][0], &c[0][1],  &c[1][0], &c[1][1],
  &v1[0], &v1[1],  &v2[0],   &v2[1],    &v3[0],   &v3[1], &ehiv,
  &r1[0], &r1[1],  &r2[0],   &r2[1],    &r3[0],   &r3[1],
  &r4[0], &r4[1],  &r5[0],   &r5[1],    &r6[0],   &r6[1],
  &r7[0], &r7[1],  &r8[0],   &r8[1],    &df,      &runid,
  &d1uk20[0], &d2uk20[0], &d3uk20[0], &md,&mi,
  &pmale[0], &randseq, 0 };
```

```
include "service.c"
```