TRANSPORTATION ANALYSIS SIMULATION SYSTEM (TRANSIMS)

Version: TRANSIMS-LANL-1.0

VOLUME 3 – FILES

28 May 1999

LA-UR 99-2579

COPYRIGHT, 1999, THE REGENTS OF THE UNIVERSITY OF CALIFORNIA. THIS SOFTWARE WAS PRODUCED UNDER A U.S. GOVERNMENT CONTRACT (W-7405-ENG-36) BY LOS ALAMOS NATIONAL LABORATORY, WHICH IS OPERATED BY THE UNIVERSITY OF CALIFORNIA FOR THE U.S. DEPARTMENT OF ENERGY. THE U.S. GOVERNMENT IS LICENSED TO USE, REPRODUCE, AND DISTRIBUTE THIS SOFTWARE. NEITHER THE GOVERNMENT NOR THE UNIVERSITY MAKES ANY WARRANTY, EXPRESS OR IMPLIED, OR ASSUMES ANY LIABILITY OR RESPONSIBILITY FOR THE USE OF THIS SOFTWARE.

TRANSIMS

Version: TRANSIMS-LANL-1.0

VOLUME 3 – FILES

28 May 1999

LA-UR 99-2579

The following persons contributed to this document:
C. L. Barrett*
R. J. Beckman*
K. P. Berkbigler*
K. R. Bisset*
B. W. Bush*
S. Eubank*
J. M. Hurford*
G. Konjevod*
D. A. Kubicek*
M. V. Marathe*
J. D. Morgeson*
M. Rickert*
P. R. Romero*
L. L. Smith*
M. P. Speckman**
P. L. Speckman**
P. E. Stretz*
G. L. Thayer*
M. D. Williams*

* Los Alamos National Laboratory, Los Alamos, NM 87545
** National Institute of Statistical Sciences, Research Triangle Park, NC

Acknowledgments

This work was supported by the U. S. Department of Transportation (Assistant Secretary for Transportation Policy, Federal Highway Administration, Federal Transit Administration), the U. S. Environmental Protection Agency, and the U. S. Department of Energy as part of the Travel Model Improvement Program.

CONTENTS

1. INTRODUCTION 8

2. SYNTHETIC POPULATION 9

2.1 TERMS 9
2.2 FILE FORMAT 9
2.3 INTERFACE FUNCTIONS 11
2.4 DATA STRUCTURES 12
2.5 FILES 13
2.6 CONFIGURATION KEYS 14
2.7 EXAMPLES 15

3. ACTIVITIES 16

3.1 TERMS 16
3.2 FILE FORMAT 16
3.3 INTERFACE FUNCTIONS 18
3.4 DATA STRUCTURES 20
3.5 FILES 22
3.6 CONFIGURATION KEYS 23
3.7 EXAMPLES 24

4. VEHICLE 25

4.1 TERMS 25
4.2 FILE FORMAT 25
4.3 INTERFACE FUNCTIONS 26
4.4 DATA STRUCTURES 27
4.5 FILES 28
4.6 EXAMPLES 28

5. PLAN 30

5.1 TERMS 30
5.2 FILE FORMAT 30
5.3 INTERFACE FUNCTIONS 34
5.4 DATA STRUCTURES 34
5.5 UTILITY PROGRAMS 35
5.6 FILES 38
5.7 CONFIGURATION KEYS 39
5.8 EXAMPLES 40

6. TRANSIT 41

6.1 TERMS 41
6.2 FILE FORMAT 41
6.3 INTERFACE FUNCTIONS 42
6.4 DATA STRUCTURES 43
6.5 FILES 43
6.6 CONFIGURATION KEYS 44

6.7 EXAMPLES 44

7. NETWORK 45

7.1 TERMS 45
7.2 FILE FORMAT 47
7.3 INTERFACE FUNCTIONS 69
7.4 DATA STRUCTURES 80
7.5 UTILITY PROGRAMS 91
7.6 FILES 92
7.7 CONFIGURATION KEYS 92
7.8 EXAMPLES 93

8. SIMULATION OUTPUT 105

8.1 TERMS 105
8.2 FILE FORMAT 105
8.3 OUTPUT FILTERING 113
8.4 INTERFACE FUNCTIONS 114
8.5 DATA STRUCTURES 124
8.6 UTILITY PROGRAMS 131
8.7 FILES 132
8.8 CONFIGURATION KEYS 133
8.9 EXAMPLES 137

9. EMISSIONS ESTIMATOR 148

9.1 TERMS 148
9.2 FILE FORMAT 148
9.3 UTILITY PROGRAMS 153
9.4 FILES 154
9.5 EXAMPLES 154

10. ITERATION DATABASE 162

10.1 TERMS 162
10.2 FILE FORMAT 162
10.3 INTERFACE FUNCTIONS 162
10.4 DATA STRUCTURES 169
10.5 UTILITY PROGRAMS 170
10.6 FILES 170

11. INDEXING 171

11.1 TERMS 171
11.2 USAGE 171
11.3 INTERFACE FUNCTIONS 173
11.4 DATA STRUCTURES 178
11.5 UTILITY PROGRAMS 181
11.6 FILES 183
11.7 EXAMPLES 183

12. VISUALIZATION 185

12.1 TERMS 185
12.2 FILE FORMAT 185
12.3 UTILITY PROGRAMS 186
12.4 FILES 187

13. CONFIGURATION 188

13.1 TERMS 188
13.2 FILE FORMAT 188
13.3 INTERFACE FUNCTIONS 188
13.4 DATA STRUCTURES 189
13.5 UTILITY PROGRAMS 189
13.6 FILES 189
13.7 CONFIGURATION KEYS 190
13.8 EXAMPLES 190

14. LOGGING 197

14.1 TERMS 197
14.2 INTERFACE FUNCTIONS 197
14.3 FILES 198
14.4 EXAMPLES 198

15. REFERENCES 199

1. INTRODUCTION

The Transportation Analysis and SIMulation System (TRANSIMS) is sponsored by the U.S. Department of Transportation, the Environmental Protection Agency, and the U.S. Department of Energy. Los Alamos National Laboratory is leading this major effort to develop new, integrated transportation and air quality forecasting procedures necessary to satisfy the Intermodal Surface Transportation Efficiency Act and the Clean Air Act and its amendments.

This document provides specifications for the complete set of files used by TRANSIMS, descriptions of the C programming language interface functions* used to read and write these files, and examples of the files. Figure 1 below shows the major files used in TRANSIMS and their relationship with the TRANSIMS software modules. All except the iteration database and index files are in standard ISO text format. The C interface libraries provide functions for reading and writing the files, and data structures in which records from the files can be stored in memory.

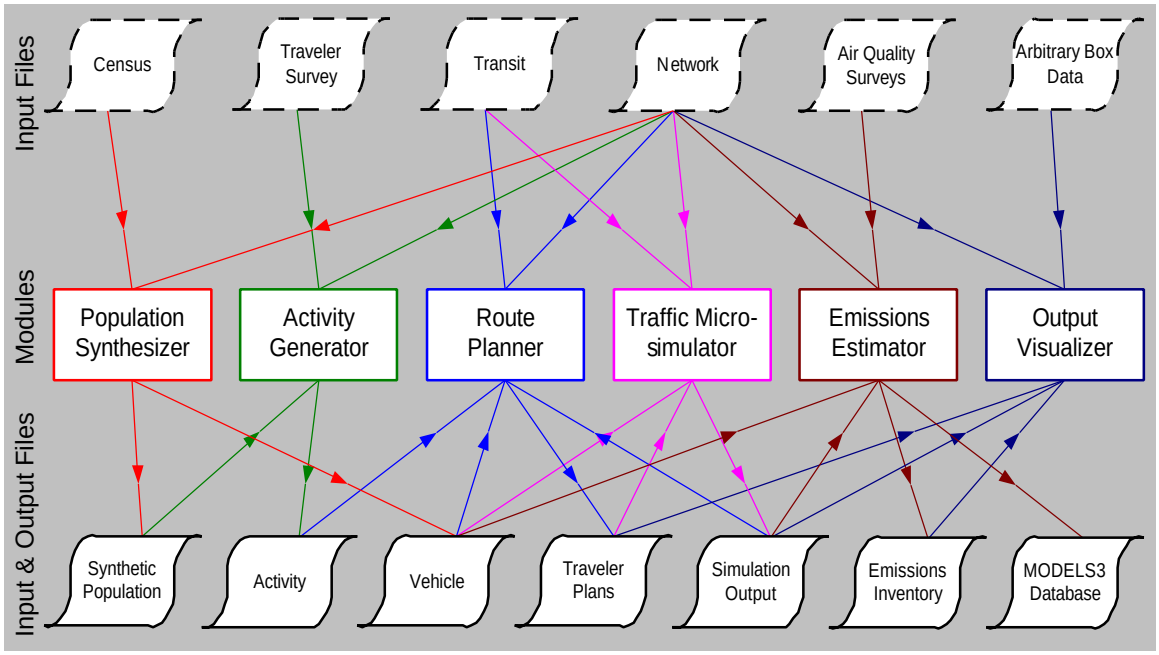

Figure 1: Interrelationship between TRANSIMS data files and software modules. The files appearing in the top row are only used as input by software modules whereas the files appearing in the bottom row are output by one module and input by one or more other modules.

NOTE: The symbol ☛ is used to indicate items that apply to the June release only.

* These functions are callable from most C++, FORTRAN, and PASCAL language implementations, too.

2. SYNTHETIC POPULATION

The TRANSIMS synthetic population system is designed to produce populations (family households, non-family households, and group quarters) that are statistically equivalent to actual populations when compared at the level of block group or higher. The methodology used by this system is described in Reference Volume 2—*Software*, Part 1—*Modules*, Section 1. The inputs to the software are U.S. Census Bureau data (STF3A and PUMS) and MABLE/GEOCORR data. Census Bureau STF3A and PUMS data formats are commonly used and are available on CD-ROM from the Census Bureau—these data inputs will not be described in any detail in this document. MABLE/GEOCORR data is relatively new, and is described in Reference Volume 2—*Software*, Part 1—*Modules*, Section 1.1.

2.1 Terms

Population	Persons grouped in households.
Demographics	Characteristics of a household or person.
Tract	U.S. Census Bureau tract number.
Block Group	U.S. Census Bureau block group number.
PUMS	U.S. Census Bureau Public Use Microdata Sample.
PUMS Household ID	The PUMS household ID number from which the synthetic population was derived.
TRANSIMS ID	Unique number assigned to each household and person. Must be greater than zero.
Home Location	The home location of the household and all persons in the household. This number is the ID of a TRANSIMS activity location and is unique for each TRANSIMS transportation network.

2.2 File Format

The synthetic population file contains two header lines, followed by the data lines.

2.2.1 Header Lines

The first line of the file contains the household demographic and user data information. The second line of the file contains person demographic information.

2.2.1.1 Format

The format of the lines is:

```
<text>: <description demog/data1> ... <description demog/dataN>
```

The `<text>` entry may be any text comment that is meaningful to the user. The `<text>` entry MUST be followed by a colon (`:`). A single word description of each of the optional household demographics in the file follows the colon. Each optional household data item that is present in the household data lines of the file must have a single word description in the household header line (line 1 of the file). Each person demographic that is present in the person data lines of the file must have a single word description in the person demographic line (line 2 of the file). The single word description must not contain white space.

2.2.1.2 Example

```
Household Demographics: PUMSHH R18UNDR RWRKR89 RHHINC
Person Demographics: AGE RELAT1 SEX WORK89
```

The household data lines in a file with this header will have four optional household demographic values (PUMSHH, R18UNDR, RWRKR89, and RHHINC). The person data lines in a file with this header will have four person demographic values (AGE, RELAT1, SEX, and WORK89).

2.2.2 Data Lines

The data for a single synthetic household span multiple lines of the synthetic population file.

2.2.2.1 Format

The first line of a household record contains the household data:

```
<TRACT ID> <Blck Grp ID> H <TRANSIMS HH ID> <# persons> <# vehicles> <Home location> [<HHData1> ... <HHData2>]
```

<TRACT ID> and <Blck Grp ID> are the census tract and block group numbers. The tract numbers are represented as an integer with the following characteristics. Tract number 1 or 1.00 is represented as 000100. Tract number 1.01 is represented as 000101. The block groups retain their integer value.

The home location is the ID of the home activity location on a TRANSIMS network. A value of -1 may be used if the home location is not known yet. Every household must eventually be assigned a home location of a TRANSIMS activity location before using TRANSIMS modules.

Following the household data are N lines, where N = number of persons in the household, of person data.

```
<TRANSIMS HH ID> P <TRANSIMS Person ID> [PersonDemog1> ... <PersonDemogN>]
```

The household and person demographics/data in the file, both number and type of the data, depend on the demographics and data used to generate the population.

2.2.2.2 Example

Household 1000 with four persons, two autos, home location of 1253, and demographics of PUMSHH (17643), R18UNDR (1), RWRKR89 (3), and RHHINC (38800). Person demographics for each member of the household are AGE, RELAT1, SEX, and WORK89.

```
Household Demographics:  PUMSHH R18UNDR RWRKR89 RHHINC
Person Demographics:     AGE RELAT1 SEX WORK89
```

```
00001 00002 H 1000 4 2 1253 17643 1 3 38800
1000 P 101 38 0 0 1
1000 P 102 36 1 1 1
1000 P 103 7 2 1 0
1000 P 104 4 2 1 0
```

2.3 Interface Functions

The synthetic population subsystem has C structures and utility functions that are used to read and write synthetic population data from TRANSIMS synthetic population files.

The function **`getNextSyntheticHH()`** reads a synthetic household from the population file. The function stores the information in a static data structure (`SyntheticHHData`) and returns a pointer to the static data. The `SyntheticHHData` structure cannot be modified by the calling program. The data should be copied if it needs to be changed. The functions **`writeSyntheticPopHeader()`** and **`writeSyntheticHH()`** are used to create a TRANSIMS synthetic population file.

2.3.1 moreSyntheticHH

Signature: `int` **`moreSyntheticHH`**`(FILE* const fp)`

Description: Boolean function used to control iteration through the synthetic population file.

Argument: `fp` – `FILE*` for the synthetic population file that must be open for reading.

Return Value: 1 if not at end of synthetic population file.
0 if EOF has been reached.

2.3.2 getNextSyntheticHH

Signature: `const SyntheticHHData*` **`getNextSyntheticHH`**`(FILE* const fp)`

Description: Reads a synthetic household from the synthetic population file. Parses and converts the values from the file and stores them in the static `SyntheticHHData` structure.

Argument: `fp` – `FILE*` for the synthetic population file that must be open for reading.

Return Value: The address of a static `SyntheticHHData` structure containing the data read from the file. Returns `NULL` on error.

2.3.3 writeSyntheticPopHeader

Signature: `int` **`writeSyntheticPopHeader`**`(FILE* const fp, char* hh_header, char* p_header)`

Description: Writes the header lines in the synthetic population file.
The format of the line is:
`<text>: <demog1> <demog2> … <demogN>`

Example:
`Household Demographics: PUMSHH R18UNDR RWRKR89 RHHINC`
`Person Demographics: AGE RELAT1 SEX WORK89`

Argument: `fp` – pointer to synthetic population file that must be open for writing with the file pointer positioned at the beginning of the file.
`hh_header` – string containing the household header information.
`p_header` – string containing the person header information.

Return Value: 1 on success.
0 on error.

2.3.4 writeSyntheticHH

Signature: `int writeSyntheticHH(FILE* const fp, const SyntheticHHData* hh)`

Description: Writes the given `SyntheticHHData` into the given synthetic population file.

Argument: `fp` – `FILE*` to the synthetic population file that must be open for writing.
`data` – address of a `SyntheticHHData` structure containing the data to be written.

Return Value: 1 on success.
0 on error.

2.4 Data Structures

2.4.1 SyntheticPersonData

This structure is used to hold synthetic person information.

```
typedef struct synPersonData_s
{
/** TRANSIMS Person ID. **/
INT32 fPersonID;

/** Array of person demographic information. **/
INT32 *fPersonDemographics;

} SyntheticPersonData;
```

2.4.2 SyntheticHHData

This structure is used to hold synthetic household information.

```
typedef struct synHHdata_s
{

/** The Census Tract ID of the household. **/
INT32 fTract;
```

```
/** The Block group ID of the household. **/
INT32 fBlockGroupID;

/** The TRANSIMS Household ID. **/
INT32 fHHId;

/** The number of persons in the household. **/
int fNumberPersons;

/** The number of vehicles owned by the household. **/
int fNumberVehicles;

/** The home location of the household – a TRANSIMS activity
 *  location ID.
 */
INT32 fHomeLocation;

/** Number of data items in the household demographics/data array. **/
int fNumberHHDemographics;

/** Array of household demographic/data information. **/
INT32 *fHHDemographics;

/** Number of demographics in the person demographics array. **/
int fNumberPersonDemographics;

/** Array of synthetic person records, one for each member of the
 *  household.
 *  The number of valid entries in this array is given by
 *  the fNumberPersons field.
 */
SyntheticPersonData *fPersons;

} SyntheticHHData;
```

2.5 Files

Table 1: Synthetic population library files.

Type	File Name	Description
Binary Files	libTIO.a	TRANSIMS Interfaces library
Source Files	synpopio.h	Defines synthetic population data structures and interface functions
	synpopio.c	Synthetic population interface functions source file

2.6 Configuration Keys

Table 2 below lists the TRANSIMS configuration file keys that specify the location of synthetic population data files.

Table 2: Synthetic population file configuration keys.

Configuration Key	Description
POP_NUMBER_HH	The number of households to be generated.
POP_BASELINE_FILE	Baseline synthetic population files—not located on a transportation network.
POP_LOCATED_FILE	Synthetic population file containing population located on a specific transportation network.
POP_STARTING_VEHICLE_ID	The first vehicle ID to be assigned. Other vehicles will be numbered sequentially from this starting number.
POP_STARTING_HH_ID	The first household ID to be assigned. Other synthetic households will be numbered sequentially from this number.
POP_STARTING_PERSON_ID	The first person ID to be assigned. Other synthetic persons will be numbered sequentially from this number.

2.7 Examples

This example program reads a synthetic population file, then writes the synthetic population data to an output file.

```
#include <stdio.h>
#include <string.h>
#include <IO/synpopio.h>

main(int argc, char *argv[])
{
   FILE *infp;
   FILE *outfp;
   const SyntheticHHData *h = NULL;
   char *hdr1 = "Household Demographics: PUMS PUMSHHID RWRKR89 RHHINC";
   char *hdr2 = "Person Demographics: AGE SEX WORK89";
   int count = 0;

   if (argc < 3) {
      fprintf(stdout, "Usage: prog <input file> < output file>\n");
      exit(0);
   }

   infp = fopen(argv[1], "r");
   if (!infp) {
      fprintf(stdout, "Failed to open input file %s\n", argv[1]);
      exit(0);”
   }

   outfp = fopen(argv[2], “w”);
   if (!outfp) {
      fprintf(stdout, “Failed to open output file %s\n”, argv[2]);
      exit(0);
   }
   writeSyntheticPopHeader(outfp, hdr1, hdr2);

   while (moreSyntheticHH(infp)) {
      h = getNextSyntheticHH(infp);
      count++;
      writeSyntheticHH(outfp, h);
   }
   fprintf(stdout, “Read/Wrote %d households\n”, count);
   fclose(infp);
   fclose(outfp);
}
```

3. ACTIVITIES

This section gives the protocol for the interaction of the TRANSIMS activity sets with the TRANSIMS planner and microsimulation.

3.1 Terms

Activity Something that a person in a household does. Each activity has parameters associated with it including priority, location, starting time, ending time, and duration.

Household One or more persons with a common home location.

Location A TRANSIMS Network Activity Location.

Mode The mode type of the transportation between activities: i.e., car, bus, walk.

3.2 File Format

A population is assumed to be in place. Each household in this population has a location on the TRANSIMS network and a unique household ID. Each person in the household also has a unique ID. A base set of activities is generated for each household in the population. These activities are modified by feedback from both the planner and the microsimulation.

The activity file for the base set of activities is an ASCII file containing the activity data. Activities for a household are grouped sequentially in the activity file. Each line of the file contains tab-delimited data fields for a single activity. Table 3 defines the meaning and format of the activity data fields. For most fields, the entry *–1* denotes an unspecified value.

The reference time is taken as 0.00 (midnight of the first day). All times are decimal numbers that denote the number of hours from 0.00. Note that each time should be given to a minimum of two decimal places to capture minutes and four decimal places if seconds are necessary. Each activity has a start time, end time, and duration range. The preferred time for each of these is given in terms of the two parameters of a beta distribution, $f(t) = C(t-L)^{a-1}(U-t)^{b-1}$ where, C is a constant, L is the lower bound of the time, U is the upper bound, and a and b are the parameters that specify the distribution. The mean of the distribution is $\frac{a}{a+b}$; $a=1$ and $b=1$ gives a uniform distribution between L and U, and the larger a and b are, the more peaked the distribution.

Table 3: Activity data definitions and format.

Field	Description	Allowed Values
Household ID	Each household has a unique household ID. Each Group Quarters is given one household ID. These numbers are assigned in the population file.	integer
PersonID	Each person is given a unique ID in the population file.	integer
Activity Type	Two types are fixed and should always have these values: Home = 1, Work = 2. Definition of other activity types may vary. Meaning of the integer value must be specified for each activity set. ☛ Activity Types generated by the NISS Activity Generator have the following meanings: 0 = home, 1 = work, 2 = shop, 3 = school, 4= visit, 5 = other.	integer: 1 through n: Example: 1 = Home 2 = Work 3 = Shop 4 = School 5 = Other
Activity Priority	A 0 is an activity of lowest priority; a priority of 9 means the activity must be done.	integer: 0 - 9
Starting Time Lower Bound	Earliest time the activity can start.	decimal
Starting Time Upper Bound	Latest time an activity can start.	decimal
Preferred Starting Time *a* parameter	The time the router will use as the best guess for the starting time. If this number is *–1*, the average of the upper and lower bounds is used.	decimal
Preferred Starting Time *b* parameter	The time the router will use as the best guess for the starting time. If this number is *–1*, the average of the upper and lower bounds is used.	decimal
Ending Time Lower Bound	The earliest time the activity can end.	decimal
Ending Time Upper Bound	The latest time the activity can end.	decimal
Preferred Ending Time *a* parameter	The time the router will use as the best guess for the activity ending time. If this number is *–1*, the average of the lower and upper bounds is used.	decimal
Preferred Ending Time *b* parameter	The time the router will use as the best guess for the activity ending time. If this number is *–1*, the average of the lower and upper bounds is used.	decimal
Duration Lower Bound	Shortest length of the activity.	decimal
Duration Upper Bound	Longest length of the activity.	decimal
Duration *a* parameter	The router will use this as the best guess of the activity duration. If this number is *–1*, the average of the upper and lower bound is used.	decimal
Duration *b* parameter	The router will use this as the best guess of the activity duration. If this number is *–1*, the average of the upper and lower bound is used.	decimal
Mode Preference for Arriving at the Activity	This number represents a grammar string that defines the mode preference to the route planner (*wcw, wt, …*). The correspondence between integer values and possible grammar strings is contained in an external file. The file defines special modes for passengers in a private auto as well as activities where no travel is done (start and end at the same location).	integer

Field	Description	Allowed Values
Vehicle ID	The vehicle ID for all activities with a mode preference of private auto, either as driver or as passenger. This field should be set to *–1* for all other modes.	integer
Number of Possible Locations for Activity	The number of possible locations in the List of Locations field if value is 1 or greater. The value 0 is not allowed. If this field is *–1*, the single value in the List of Locations field is an index into a group of activities.	-1, integer ≥ 1
List of Activity Locations	If the Number of Possible Locations field is 1 or greater, this field contains a list of activity location IDs where an activity may take place. If the Number of Possible Locations field is *–1*, this field contains a number that is an index into a group of activities.	integer [integer] …
Number of Other Participants in the Activity	The number of others in the population who might participate and use the same transportation (e.g., the same car). The number is 0 if the person is to travel alone to the activity.	integer
List of Other Participants	Person IDs of other participants using the same transportation. This field should be present only when the value of the Number of Other Participants field is > 0. If this person is the driver of the car, this list contains the person IDs of the other passengers in the car. If this person is a passenger in the car, this list contains the person ID of the driver.	[integer] [integer] …
Activity Group Number	Every activity for an individual will have a number. Sets of activities that must be done together will have the same number.	integer

3.3 Interface Functions

The activity subsystem has C structures and utility functions that are used to read and write activity data from a TRANSIMS activity file. These functions assume that all of the activities for a household are grouped sequentially in the TRANSIMS activity file.

The functions **`getNextActivity()`** and **`getNextHousehold()`** read an activity/household from an activity file in ASCII format. The functions store the information in static data structures (`ActivityData`) and return a pointer to the static data. The `ActivityData` structures or arrays cannot be modified by the calling program. The data should be copied if it needs to be changed.

The functions **`writeActivity()`** and **`writeHousehold()`** accept the `ActivityData` structures containing the information to be written as arguments.

The read functions provide a mechanism for iterating through the activity file reading either individual activities or the activities for a household. The write functions can write a single activity or a household's activities to the file.

3.3.1 moreActivities

Signature: `int moreActivities(FILE * const fp)`

Description: Boolean function used to control iteration through the activity file.

Argument: `fp – FILE *` for the activity file, which must be open for reading.

Return Value: 1 if not at end of activity file
0 if EOF has been reached

3.3.2 getNextActivity

Signature: `const ActivityData * getNextActivity(FILE * const)`

Description: Reads an activity from the activity file. Parses and converts the string values from the file and stores them in a static `ActivityData` structure. Allocates storage for the `fOtherParticipantsList` and `fLocations` arrays based on data in the file.

Argument: `fp – FILE *` to the activity, which must be open for reading.

Return Value: The address of an unmodifiable `ActivityData` structure containing the activity data from the file. Returns `NULL` on error.

3.3.3 getNextHousehold

Signature: `const ActivityData * getNextHousehold(FILE * const fp, int* arraySize)`

Description: Reads the activities for a household from the activity file.
Constructs an `ActivityData` structure for each activity in the household. Parses the activities and stores them in an array of `ActivityData` structures.

Argument: `fp – FILE *` to the activity file, which must be open for reading.

Return Value: An array of unmodifiable `ActivityData` structures that contains the activity data for the household. Returns `NULL` on error. The number of activities for the household is returned in the `arraySize` argument.

3.3.4 writeActivity

Signature: `int writeActivity(FILE * const fp, const ActivityData * data)`

Description: Writes the given `ActivityData` into a line of the given activity file.

Argument: `fp – FILE *` to the activity file, which must be open for writing

`data` – address of an `ActivityData` structure containing the data to be written.

Return Value: 1 on success.
0 on error.

3.3.5 writeHousehold

Signature:
```
int writeHousehold(FILE * fp, ActivityData
* data, int arraySize)
```

Description: Writes the activities for a household into the given file.

Argument: `fp – FILE*` to the activity file, which must be open for writing.
`data` – address of an `ActivityData` array containing the household activity data to be written.
`arraySize` – the number of activities in the data array.

Return Value: 1 on success.
0 on error.

3.4 Data Structures

3.4.1 ActivityTimeSpec

This structure is used for activity time specifications.

```
typedef struct act_time_spec_s
{
/** The lower bound of the time interval. **/
REAL fLowerBound;

/** The upper bound of the time interval. **/
REAL fUpperBound;

/** The A parameter for the beta distribution. **/
REAL fAParameter;

/** The B parameter for the beta distribution. **/
REAL fBParameter;

} ActivityTimeSpec;
```

Each activity has a start time, end time, and duration range. The preferred time for each of these is given in terms of the two parameters of a beta distribution, $f(t)=C(t-L)^{a-1}(U-t)^{b-1}$, where C is a constant, L is the lower bound of the time, U is the upper bound and a and b are the parameters that specify the distribution. The mean of the distribution is $a/(a+b)$; $a=1$ and $b=1$ gives a uniform distribution between L and U. Larger values for a and b result in a more peaked distribution. If the a and/or b parameter is equal to -1.0, an average of the lower and upper bound will be used.

The reference time is taken as 0.00 (midnight of the first day). All times are decimal numbers that denote the number of hours from 0.00. Note that each time should be given to a minimum of two decimal places to capture minutes and four decimal places if seconds are necessary.

3.4.2 ActivityData

This structure is used to store the data for a single activity as defined by one line in the activity file.

```
typedef struct actdata_s
{
/** The household Id. **/
INT32 fHouseholdId;

/** The person Id. **/
INT32 fPersonId;

/** Activity type.  An integer value representing
 *  the activity type such as home, work, school, shopping,
 *  other, wait at transit stop, ... .
**/
INT32 fType;

/** Priority ranking of the activity in the range 0 - 9,
 *  where 0 is the lowest priority and 9 means the activity
 *  must be done.
**/
INT32 fPriority;

/** Integer value defining transportation mode used to arrive
 *  at the activity.
**/
INT32 fModePreference;

/** The ID of the vehicle to be used when the mode preference is private
 *  auto, either as a driver or passenger.  Set to –1 for all other mode
 *  preferences.
**/
INT32 fVehicleId;

/** The number of locations where the activity can take place.
 *  This field is used to provide information about the
 *  fActivityGroupIndex and fPossibleLocationsList fields.
 *  A value of 1 or greater indicates that the fPossibleLocationsList
 *  contains a list of locations for the activity.
 *  A value of -1 indicates that the fActivityGroupIndex field
 *  contains an index number into a group of activities.
**/
INT32 fPossibleLocations;

/** The number of other people that will participate in the activity
 *  and use the same transportation.  Value is 0 if the person is
 *  traveling alone to the activity.  If the value is > 0, a list
 *  of the IDs of the other participants is entered in the
 *  fOtherParticipantsList array.
**/
INT32 fOtherParticipants;
```

```
/** Number of the activity for this individual.  Every activity for
 *  an individual has a number.  Groups of activities that must be
 *  done together have the same number.
**/
INT32 fActivityGroupNumber;

/** An array of personIds for other participants in the activity
 *  that will use the same transportation.  There are no valid entries
 *  in this array if the value of fOtherParticipants is 0.
**/
INT32 *fOtherParticipantsList;

/** Index into a group of activities (integer).
 *  Used only when value of fPossibleLocations is -1.
**/
INT32 fActivityGroupIndex;

/** An array of possible locations (integer IDs) where
 *  the activity will occur. Used when value of
 *  fPossibleLocations is 1 or greater.
**/
INT32 *fLocations;

/** Preferred start time for the activity.  The ActivityTimeSpec
 *  structure contains the specification parameters for a beta
 *  distribution of the preferred time.
**/
ActivityTimeSpec fStart;

/** Preferred end time for the activity.  The ActivityTimeSpec
 *  structure contains the specification parameters for a beta
 *  distribution of the preferred time.
**/
ActivityTimeSpec fEnd;

/** Preferred duration for the activity.  The ActivityTimeSpec
 *  structure contains the specification parameters for a beta
 *  distribution of the preferred time.
**/
ActivityTimeSpec fDuration;

} ActivityData;
```

3.5 Files

Table 4: Activity library files.

Type	File Name	Description
Binary Files	libTIO.a	TRANSIMS Interfaces library
Source Files	actio.h	Defines activity data structures and interface functions
	activityio.c	Activity interface functions source file

3.6 Configuration Keys

Table 5 below lists the TRANSIMS configuration file keys that specify the location of activity data files.

Table 5: Activity file configuration keys.

Configuration Key	Description
ACT_FULL_OUTPUT	The file containing a complete activity set generated from a population.
ACT_PARTIAL_OUTPUT	The file containing activities from a partial activity generation for specified persons.
ACT_FEEDBACK_FILE	The file containing a list of travelers and associated commands for activity regeneration.
ACT_WORK_LOC_ALPHA	The alpha parameter used to generate work locations in the simplified activity generator.
ACT_WORK_LOC_BETA	The beta parameter used to generate work locations in the simplified activity generator.
ACT_WORK_LOC_GAMMA	The gamma value used to generate work locations in the simplified activity generator.
ACT_TIME_ALPHA	The alpha parameter used to generate activity times in the simplified activity generator.
ACT_TIME_BETA	The beta parameter used to generate activity times in the simplified activity generator.
ACT_MODE_ALPHA	The alpha parameter used to generate mode choice in the simplified activity generator.
ACT_MODE_BETA	The beta parameter used to generate mode choice in the simplified activity generator.
ACT_WORK_LOCATION_OPTION	The option used to select the work location algorithm in the simplified activity generator.
ACT_MODE_CHOICE_OPTION	The option used to select the mode choice algorithm in the simplified activity generator.
ACT_HOME_HEADER	The user data column header in the network activity location file used to specify single family home locations.
ACT_MULTI_FAMILY_HEADER	The user data column header in the network activity location file used to specify multifamily home locations.
ACT_WORK_HEADER	The user data column header in the network activity location file used to specify work locations.
ACT_ACCESS_HEADER	The user data column header in the network activity location file used to specify access to transit.
ACT_TRACT_HEADER	The user data column header in the network activity location file used to specify census tract.
ACT_BLOCKGROUP_HEADER	The user data column header in the network activity location file used to specify block group.

3.7 Examples

Read all of the households in the activity file, then write the activity information for the household to another file. The data for each household is stored in an array of `ActivityData` structures.

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

int main(int argc, char *argv[])
{
   FILE *fp;
   FILE *outfp;
   int number_activities;
   const ActivityData *hh;

   if (argc < 3) {
      fprintf(stdout, "Usage:  testact <activity input file> <activity output file>\n ");
      exit(0);
   }

   fp = fopen(argv[1], "r");
   if (fp == NULL) {
      printf("Failed to open file %s...exiting\n", argv[1]);
      exit(0);
   }

   outfp = fopen(argv[2], "w");
   if (outfp == NULL) {
      printf("Failed to open file %s...exiting\n", argv[2]);
      exit(0);
   }

   while (moreActivities(fp)) {
      hh = getNextHousehold(fp, &number_activities);
      if (number_activities == 0) {
         fprintf(stderr, "Failed to get household\n");
      } else {
         writeHousehold(outfp, hh, number_activities);
      }
   }

   fclose(fp);
   fclose(outfp);
   return 0;
}
```

4. VEHICLE

This section gives the protocol for the interaction of the TRANSIMS Vehicle library with the TRANSIMS planner and microsimulation. Private vehicles are generated and assigned to households by the TRANSIMS population synthesizer. The activity generator assigns a set of possible vehicles to each member of a household. Freight and transit vehicles (and the plans for their drivers) are generated by separate utilities, but must be included in the vehicle database. The vehicle IDs assigned by these utilities must be unique.

4.1 Terms

Vehicle Any driver must have an associated vehicle.

Vehicle Type Vehicles can be classified in several ways: by network type (e.g., definitions used in imposing lane use or turn prohibition restrictions); by usage (e.g., transit, private auto, carpool, jitney), which affects simulation; by performance characteristics (e.g., length, acceleration profile); by emissions type (e.g., power/weight ratio). In this section, network type is considered to be the primary type.

4.2 File Format

Fields in the vehicle file are tab- or space-delimited.

Each line of the vehicle file contains four mandatory fields:
1) household ID
2) vehicle ID
3) ID of the starting location
4) the TRANSIMS network type of the vehicle

The TRANSIMS network vehicle type must be one of the following values:
1) 1 = Auto
2) 2 = Truck
3) 4 = Taxi
4) 5 = Bus
5) 6 = Trolley
6) 7 = StreetCar
7) 8 = LightRail
8) 9 = RapidRail
9) 10 = RegionalRail

The line may contain optional integer fields whose meaning is user defined. The number of these identifier fields may vary among different vehicle files. The number of optional identifier fields must be the same on every line within a vehicle file. The value *-1* is used as a default placeholder value for both the starting location and optional integer fields when the values are unknown or unused.

Format

```
<Household ID> <Vehicle ID> <starting location> <network type> [<int 1> ... <int n>]
```

Example

Household 1460 has two vehicles (500100 and 500101); both start at the home location (78) and are of network type auto (1). Two optional user-defined integer fields are present in this file. The first field is the emissions vehicle type (10), which is the same for both vehicles. The second integer field is an indicator of the maintenance level of the vehicle. Note that the second vehicle (500101) has unknown/unused value (-1) for the second integer field.

```
1460 500100 78 1 10 30
1460 500101 78 1 10 -1
```

4.3 Interface Functions

The vehicle subsystem has C structures and utility functions that are used to read and write data from a TRANSIMS vehicle file.

The function **`getNextVehicle()`** reads vehicle data from a vehicle file in ASCII format. The function stores the information in an unmodifiable data structure (`VehicleData`), and returns a pointer to the structure. Since the `VehicleData` structure cannot be modified by the calling program, the data should be copied if it needs to be changed.

The function **`writeVehicle()`** takes a `VehicleData` structure as an argument containing the information to be written. The **`getNextVehicle()`** function combined with the **`moreVehicles()`** function provides a mechanism for iterating through the vehicle file reading the vehicle data.

4.3.1 moreVehicles

Signature: `int moreVehicles(FILE * const fp)`

Description: Boolean function used to control iteration through the vehicle file.

Argument: `fp – FILE *` for the vehicle file that must be open for reading.

Return Value: 1 if not at end of vehicle file.
0 if EOF has been reached.

4.3.2 getNextVehicle

Signature: `const VehicleData * getNextVehicle(FILE * const fp)`

Description: Reads a line of vehicle data from the vehicle file. Parses and converts the string values from the file and stores them in the static `VehicleData` structure `fVehicle`.

Argument: `fp – FILE *` to the vehicle file, which must be open for reading.

Return Value: The address of a static `VehicleData` structure containing the vehicle data read from the file. Returns `NULL` on error.

4.3.3 writeVehicle

Signature: `int writeVehicle(FILE * const fp, const VehicleData * data){`

Description: Writes the given `VehicleData` into a line of the given vehicle file.

Argument: `fp – FILE *` to the vehicle file, which must be open for writing.
`data` – address of a `VehicleData` structure containing the data to be written.

Return Value: 1 on success.
0 on error.

4.4 Data Structures

4.4.1 VehicleData

This structure is used to store the data for a single vehicle as defined by one line in the vehicle file.

```
typedef struct vehdata_s
{
/** The household Id. **/
INT32 fHouseholdId;

/** The vehicle ID. **/
INT32 fVehicleId;

/** The ID starting location of the vehicle.  –1 is used if
 *  the starting location is unknown or to indicate that the
 *  route planner should choose the starting location.
**/
INT32 fStartingLocation;

/** The TRANSIMS network vehicle type.
 *  Must be one of the following values:
 *    1 = Auto
 *    2 = Truck
 *    4 = Taxi
 *    5 = Bus
 *    6 = Trolley
 *    7 = StreetCar
 *    8 = LightRail
 *    9 = RapidRail
 *   10 = RegionalRail
 *   -1 = Unknown
**/
INT32 fNetworkVehicleType;

/** The number of values in the fIdentifiers array. **/
INT32 fNumberIdentifiers;
```

```
/** Optional array of user defined integer values.
 *  The number of entries in the array is variable
 *  but must be the same for every line of the file.
 *  If no user-defined values are present in the file,
 *  fIdentifiers will be NULL.
**/
INT32 *fIdentifiers;

} VehicleData;
```

4.5 Files

Table 6: Vehicle library files.

Type	File Name	Description
Binary Files	libTIO.a	The TRANSIMS Interfaces library
Source Files	vehio.h	Defines vehicle data structures and interface functions
	vehio.c	Vehicle interface functions source file

4.6 Examples

Read all of the data in the vehicle file then write the vehicle information to another file. The data for each vehicle is stored in a `VehicleData` structure.

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

int main(int argc, char *argv[])
{
   FILE  *fp;
   FILE  *outfp;
   int      count = 0;
   const VehicleData    *veh;

   if (argc < 3) {
      fprintf(stdout, "Usage:  testveh <veh input file> <output file>\n");
      exit(0);
   }

   fp = fopen(argv[1], "r");
   if (fp == NULL) {
      printf("Failed to open file %s...exiting\n", argv[1]);
      exit(0);
   }

   outfp = fopen(argv[2], "w");
   if (outfp == NULL) {
      printf("Failed to open file %s...exiting\n", argv[2]);
      exit(0);
   }

   while (moreVehicles(fp)) {
      veh = getNextVehicle(fp);
      if (veh == NULL) {
         fprintf(stderr, "Error FAILED to get vehicle...exiting\n");
         break;
```

```
      }
      count++;
      if (!writeVehicle(outfp, veh)) {
         fprintf(stderr, “Failed to write vehicle %d\n”, veh>fVehicleId);
      }
   }

   fclose(fp);
   fclose (outfp);
   return 0;
}
```

5. PLAN

This section gives the protocol of the TRANSIMS planner file interface with the microsimulation.

5.1 Terms

Plan A plan consists of a sequence of trips.

Trip A trip consists of a sequence of (unimodal) legs. There will be a trip between each pair of activities specified in the activity file. There will also be a trip consisting of a single non-transportation leg for each activity. Each trip starts and ends at an activity location accessory as specified in the activity file.

Leg A (unimodal) leg describes a traveler's movement through the network. A leg must start and end at an accessory. The leg contains such information as departure time, transportation mode, and route through the network.

Accessory See *Network* section below.

5.2 File Format

The TRANSIMS code supplies a library of C routines as well as a `TPlan` C++ object that can read and write this format.

The format consists of a required *header* and a set of *mode-dependent data*. The header contains information common to every kind of leg. Code that uses the plans may choose to ignore some or all of the mode-dependent data. For example, the CA microsimulation will not simulate walking or bicycling, but will use the estimated duration from the planner. Since the origin, destination, and expected duration of any leg are available in the header information, the simulation does not need any data in the mode-dependent part of a walk leg.

5.2.1 Data Definitions and Format

Table 7: Plan data definitions and format.

Column Name	Description	Allowed Values
Traveler (Person) ID	Each person is given a unique ID in the population file.	integer
User Field	Available to the user to set as desired. Its value is not used internally by the simulation, but is passed to the output system for use in filtering.	integer
Trip ID	Numbers the trips for the traveler sequentially from 1. The trip ID is not used by the simulation.	unsigned 16-bit integer
Leg ID	Numbers the legs within a trip sequentially from 1.	integer
First Leg Flag	Not used.	boolean
Last Leg Flag	Not used.	boolean
Activation Time	The earliest time the simulation needs to worry about this leg. It is generally the starting time (estimated by the planner) for a leg. For a transit leg, however, it represents the arrival time of the passenger at the transit stop, rather than the arrival time of the transit vehicle.	integer: seconds since midnight
Start Accessory ID	Denotes the network accessory ID of the starting location for this leg.	unsigned long
Start Accessory Type	Denotes the type of accessory of the corresponding location. It is necessary because the IDs are not globally unique over accessories. It should be one of: 1) activity location 2) parking 3) transit stop as defined in `TNetAccessory::EType` of *NET/Accessory.h*.	integer enumeration
End Accessory ID	As above, except it is for the destination rather than the starting accessory.	unsigned long, integer
End Accessory Type	As above, except it is for the destination rather than the starting accessory.	unsigned long, integer
Duration	In conjunction with Stop Time and Max Time Flag, specifies how long this leg is expected to take.	integer: seconds
Stop Time	In conjunction with Stop Time and Max Time Flag, specifies an absolute ending time for this leg.	integer: seconds since midnight
Max Time Flag	If true, the end of this activity is best estimated as `max(start time + duration; stop_time)`. Otherwise, use the minimum instead. In the simulation, the actual start time is used, rather than the estimated activation time.	boolean
Driver Flag	True, if the traveler is driving a vehicle on this leg.	boolean
Mode	Mode of travel. This, together with the driver flag, determines the interpretation of the mode-dependent data following the header. Currently, it must be one of: 0 - car 1 - transit 2 - pedestrian	integer, enumeration

Column Name	Description	Allowed Values
	3 - bicycle 4 - non-transportation activity as defined in the `TPlan::ETravelMode enum` of *PLAN/Plan.h*.	
Vehicle Type	Type of vehicle. Currently, it must be one of: 0 - walk 1 – auto 2 – truck 3 – bicycle 4 – taxi 5 – bus 6 – trolley 7 – street car 8 – light rail 9 – rapid rail 10 – regional rail	integer, enumeration
Number of Tokens	Number of white-space-separated tokens in the mode-dependent data block (not including num_tokens itself).	integer

The plan file contains a series of records, each of which specifies a single leg of a traveler's trip. Each record contains the fields shown in the table above, in the order shown, separated by white space (space(s), tab(s), and/or a single newline). The field names are not written in the data file. There is a blank line separating each pair of records. The file is written in ASCII text. Efficiency concerns are addressed by accessing plan files through an index. See the *Index* section for details.

The combination of Duration, Stop Time, and Max Time Flag allows flexible specification of departure times. For example, attending a movie might be encoded as:

```
duration = 0 seconds;
stop time = 20*3600 + 30*60 = 73800;
maxTime = true;
```

which means, "this activity ends at 8:30 p.m., or as soon as the traveler arrives, whichever is later." Similarly, *work* might be encoded as:

```
duration = 8 hours;
stop time = 17*3600 = 61200;
maxTime = true;
```

which means "stay at work until 5:00 p.m., or eight hours after arrival, whichever is later." Shopping at lunch might be encoded as:

```
duration = 0.5 hours;
stop time = 12*3600 + 45*60 = 45900;
maxTime = false;
```

which means "shop for half an hour or until 12:45 p.m., whichever is earlier."

5.2.2 Mode-dependent Data

Mode-dependent data is written by the TRANSIMS router and interpreted by the TRANSIMS microsimulation.

Table 8: Mode-dependent data for a car driver.

Data	Description	Allowed Values
Vehicle ID	Each vehicle (with its ID) available in the simulation is listed in the vehicle database.	integer
Number of Passengers	The number of passengers, not including the driver, on this leg.	integer
List of Node IDs	The nodes (in order) through which the driver's route will pass.	integer
List of Passenger IDs	The traveler ID of each passenger to be carried on this leg.	integer

Table 9: Mode-dependent data for a car passenger.

Data	Description	Allowed Values
Vehicle ID	Each vehicle (with its ID) available in the simulation is listed in the vehicle database.	integer

Table 10: Mode-dependent data for a transit driver.

Data	Description	Allowed Values
Vehicle ID	Each vehicle (with its ID) available in the simulation is listed in the vehicle database.	integer
Route ID	Route IDs are specified in the transit route file. Only one route ID is allowed per leg.	integer
List of Node IDs	The nodes (in order) through which the driver's route will pass.	integer

Table 11: Mode-dependent data for a transit passenger.

Data	Description	Allowed Values
Route ID	Traveler will board any transit vehicle whose driver's plan matches this Route ID.	integer

Table 12: Mode-dependent data for a pedestrian.

Data	Description	Allowed Values
List of Node IDs	The nodes (in order) through which the traveler's route will pass.	integer

For activity legs, there is no mode-dependent data.

5.3 Interface Functions

The plan subsystem has C structures and utility functions that are used to read and write plan data from a TRANSIMS plan file. The plans are stored in ASCII format in a plan file. The majority of the time required for reading a plan from disk lies in converting the ASCII representation of numbers into another format. Thus, for efficiency in those cases where only one or two fields of a plan record are required, we provide routines that read the record in as an ASCII string and allow conversion of particular fields from the string, as well as routines that read the string and convert every field.

The function **`getNextLeg()`** reads a plan record from a plan file in ASCII format. The functions store the information in static data structures (`LegData`) and return a pointer to the static data. The `LegData` structures or arrays cannot be modified by the calling program. The data should be copied if it needs to be changed.

The function **`writeLeg()`** accepts the `LegData` structures containing the information to be written as arguments. The read functions provide a mechanism for iterating through the plan file. The write functions can write a single plan record to the file.

5.3.1 getNextLeg

Signature: `const LegData * const` **`getNextLeg`**`(FILE * const fp)`

Description: Reads a leg from the leg file. Parses and converts the string values from the file and stores them in the static `LegData` structure `fLeg`. Allocates storage for the `fData` array based on data in the file.

Argument: fp – `FILE *` to the leg file, which must be open for writing.

Return Value: The address of a `LegData` structure containing the leg data read from the file. Returns `NULL` on error.

5.4 Data Structures

5.4.1 LegData

This structure is used for modal leg data.

```
typedef struct plandata_s
{
/** The TravID field. **/
UNIT32 fTravId;

/** The User field. **/
INT32 fUser;

/** The fTrip field. **/
INT32 fTrip;

/** The Leg field. **/
```

```
INT32 fLeg;

/** The FirstLeg field. **/
INT32 fFirstLeg;

/** The LastLeg field. **/
INT32 fLastLeg;

/** The ActivationTime field. **/
INT32 fActivationTime;

/** The StartAcc field. **/
INT32 fStartArc;

/** The StartAccType field. **/
INT32 fStartAccType;

/** The EndAcc field. **/
INT32 fEndAcc;

/** The EndAccType field. **/
INT32 fEndAccType;

/** The Duration field. **/
INT32 fDuration;

/** The StopTime field. **/
INT32 fStopTime;

/** The MaxTime field. **/
INT32 fMaxTime;

/** The DriverFlag field. **/
INT32 fDriverFlag;

/** The Mode flag. **/
INT32 fMode;

/** The VehicleType field. **/
INT32 fVehicleType;

} LegData;
```

5.5 Utility Programs

5.5.1 PlanFilter

PlanFilter provides sorting, merging, selection and validation of plans. It constructs two indexes for each input and output plan file it touches, one sorted by time and the other by traveler. Currently existing indexes are used if they are up-to-date. If the *-v* option is used, only valid plan sequences are included in the indexes, and an index of invalid plans is built. All times are measured in seconds since midnight.

Usage:

```
PlanFilter [-h] [-d] [-f] [-w] [-v netConfigFile] [-s startTime] [-e endTime] [-t travId]*
[-r <travFile>] [-o <outFile>] <planFile>*
```

where:

h = print this message
d = defragment the file: create a new plan file containing the merged, filtered plans; the *-o* flag must accompany this flag
f = sort output by traveler
v = validate each trip chain: `netConfigFile` must be a TRANSIMS configuration file specifying a network database (Validation may be time-consuming.)
s = include only legs whose (estimated) departure time is >= `startTime`
e = include only legs whose (estimated) arrival time is <= `endTime`
t = include only legs for traveler `travId`; implies the *-f* flag (May appear an arbitrary number of times.)
r = include only legs for travelers specified in `travFile`; implies the *-f* option (May appear together with the *-t* options.)
o = place output in `outFile`; default is standard output

Arguments that do not start with "-" are assumed to be input plan files.

5.5.2 DistributePlan

5.5.2.1 Overview

The purpose of this tool is to create a separate pair of indexes into a plan file for each processor in a multiprocessor run of the microsimulation. Each leg of a trip is assigned to the processor that has responsibility for the starting accessory of that leg. This allows the processors to get travelers into the simulation more efficiently than if each processor had to read in every leg, discarding those that it did not need.

5.5.2.2 Algorithm

DistributePlans uses a mapping from accessory type and ID to CPU number. This mapping, or partition, is created during a simulation run as specified by the values of certain configuration file keys. It is saved in a file specified by the PAR_PARTITION_FILE key if the PAR_SAVE_PARTITION key is set. Note that, if run time information is saved during the simulation (using the PAR_RTM_INPUT_FILE) and that information is used to partition the network on the next run (by setting the CA_USE_RTM_FEEDBACK key), the partition can change from one run to the next.

DistributePlans can also generate the partition if none is present. In this case, the partition can be saved and used by the microsimulation (by turning off both the PAR_USE_METIS_PARTITION and PAR_USE_OB_PARTITION keys).

DistributePlans creates an index file for each processor in the partition, using a simple naming convention that allows the individual slaves to find the correct index file if it exists.

For each leg in a plan file specified by the PLAN_FILE configuration key, *DistributePlans* determines the starting location's accessory type and ID. Next, it finds the processor number

assigned responsibility for that location. Finally, it places an index entry for the leg in the file for that processor. The underlying data is not moved.

There is one additional task handled by *DistributePlans*. When a trip's legs are distributed, it becomes difficult for any processor to know whether a particular leg represents the first or last leg a traveler will undertake during the course of the simulation. This information is required because on a traveler's first leg, the associated object must be created within the simulation. On all other legs, the traveler object must not be created—instead the simulation must wait for the traveler object to arrive at that leg's starting location before allowing it to continue. Similarly, but not quite as importantly, efficient use of memory requires deleting the traveler object at the end of it's last leg.

DistributePlans ensures that the appropriate information about each traveler is made available to the simulation. It places an index entry for the first leg of each traveler's trip into each distributed index. This, in combination with the ability of the microsimulation to use both a traveler ID sorted index and a time sorted index allows it to correctly create and destroy travelers.

5.5.2.3 Usage

```
DistributePlans <config-file>
```

5.5.2.4 Configuration Keys

The keys listed in Table 13 are used when a partition already exists.

Table 13: Keys if a partition exists.

Configuration Key	Description
PAR_PARTITION_FILE	Name of a file providing a mapping from nodes to processors. This file also includes node coordinates, so it can be used to display the partition.
PLAN_FILE	The name of a plan file to distribute over the partition.
NET_*	The configuration file should also contain all the NET_ keys.

The keys listed in Table 14 are used to generate a partition if one does not already exist.

Table 14: Keys to generate a partition.

Configuration Key	Description
PARTITIONER_USE_NETWORK_CACHE	If set, the code will read in a binary cached version of the network.
GBL_CELL_LENGTH	The length of a CA cell in meters.
PAR_MIN_CELLS_TO_SPLIT	Splitting short links can cause problems in the dynamics of the microsimulation. No links with fewer cells than this will be split.
PAR_SLAVES	The number of processors in the partition.
PAR_RTM_PENALTY_FACTOR, PAR_RTM_INPUT_FILE, CA_USE_RTM_FEEDBACK	See the description in the software modules volume, Microsimulation section on configuration keys.
PAR_HOST_COUNT, PAR_HOST_CPUS_<n>, PAR_HOST_SPEED_<n>	These parameters are used to describe the machine environment. Relative processor speed will be taken into account when creating the partition.
PAR_USE_METIS_PARTITION, PAR_USE_OB_PARTITION	If PAR_USE_METIS_PARTITION is set, the partition will be determined using the METIS graph partitioning library. If PAR_USE_OB_PARTITION an is set, orthogonal bisection algorithm will be used. If neither is set, the partition specified in the PAR_PARTITION_FILE will be used.
PAR_SAVE_PARTITION	The partition will be saved in PAR_PARTITION_FILE only if this is set.

5.5.2.5 Troubleshooting

If a very large number of processors are used, the algorithm may run into an operating system limit on the number of open file descriptors allowed.

Distributing the indexes makes the plan-reading phase of the microsimulation more efficient. However, there may be I/O considerations that are important when a large number of processors are trying to gain access to the same underlying data files. This problem could be addressed by using the *PlanFilter* tool to create a separate data file for each of the indexes created and the *IndexPlanFile* tool to recreate the indexes, now pointing at the distributed plan files instead of a global file.

5.6 Files

Table 15: Plan library files.

Type	File Name	Description
Binary Files	libTIO.a	TRANSIMS Interfaces library.
Source Files	planio.c	Defines plan data structures and interface functions.
	planio.h	Plan interface functions source file.

5.7 Configuration Keys

Table 16. Plan file configuration keys.

Configuration Key	Description
PLAN_FILE	Location of a file containing plans, or the base name of an index which points to plan files. Used by the Route Planner for output and the microsimulation and Selector for input.
CA_USE_PARTITIONED_ROUTE_FILES	If this key is set, the simulation expects to find separate indices into a plan file for each slave. These can be produced using a partition file and the *DistributePlans* utility.

5.8 Examples

Table 17 gives a six-leg plan for traveler 1. It is a walk-car-walk-bus-walk plan.

Table 17: Annotated example of a plan.

Trip/Leg	Plan	Description

Trip 1/Leg 1	`1 156 1 1 1 0`
	`25200 123 1 456`
	`33 25200 1`
	`0 2`
	`2`
	`1000 1001`
		The user has chosen to mark this leg with the code 156, which has meaning only to that user but will be duly reported in any output concerned with this leg. It is trip 1, leg 1 for this traveler. It is the first leg to be simulated for this traveler, but not the last. The planner expects the trip to start at 25200 = 7∗3600 = 7 AM. The leg will start at activity location 123 and end at parking accessory 456. The planner expects the trip to take 33 seconds. The traveler's next leg will begin upon arrival at the destination or 33 seconds after departure from the origin, whichever is later. The traveler is not driving a vehicle and is, in fact, walking (mode = 2). There are two tokens of mode-dependent data, which in this case might be the nodes traversed. The CA microsimulation would probably simply use the planner's estimated duration and place the traveler in the destination queue 33 seconds after his arrival at the origin. However, the simulation could also choose to estimate its own duration. The microsimulation will not use the node information.

Trip 1/Leg 2	`1 156 1 2 0 0`
	`25233 456 2 789`
	`1314 0 1`
	`1 0`
	`18`
	`0 0`
	`1 2 3 4 5 6 7 8 9`
	`10 11 12 13 14 15 16`
		Leg 2 of trip 1 is neither the first nor the last. The traveler will be driving (driver flag = 1) a car (mode = 0) from parking accessory 456 to parking accessory 789 via the 16 nodes 1-16 using vehicle 0, carrying no passengers. The expected start time is 7:00:33 a.m., and the expected duration is 1314 seconds.

Trip 1/Leg 3	`1 156 1 3 0 0`
	`26547 789 2 10 2`
	`127 0 1`
	`1 0`
	`5`
	`0 1`
	`17 18`
	`1000`
		Traveler 1 picks up one passenger (traveler 1000) and drives to parking accessory 10 via nodes 17 and 18.

Trip 1/Leg 4	`1 156 1 4 0 0`
	`26674 10 2 11 3`
	`30 0 1`
	`0 2`
	`0`
		The traveler walks (mode = 2) from parking accessory 10 to bus stop (accessory type = 3) 11. The planner, knowing that the simulation will not simulate walking, has chosen not to write out the details of the path the walker will take (Number Of Tokens = 0).

Trip 1/Leg 5	`1 156 1 5 0 0`
	`26704 11 3 4 3`
	`1502 0 1`
	`0 1`
	`1`
	`72`
		The traveler will ride in (driver_flag = 0) the first bus (mode = 1) arriving on route 72, from bus stop 11 to bus stop 4.

Trip 1/Leg 6	`0 156 1 0 0`
	`28206 4 3 5 1`
	`31 0 1`
	`1 2`
	`0`
		The traveler takes 31 seconds to walk from bus stop 4 to activity location 5.

Trip 2/Leg 1	`1 156 2 0 1`
	`28237 5 1 5 1`
	`28800 61200 1`
	`1 4`
	`0`
		This is the <u>first leg</u> of trip 2 for traveler 1. Since the last leg flag is set, it is also the last leg that will be simulated. It is an activity (mode = 4) that ends at 5:00 p.m. (= 17 ∗ 3600 = 61200 seconds) or eight hours (= 8 ∗ 3600 = 28800) after arrival, whichever is later. There is no data associated with this leg, although the planner could, in principle, add anything—a list of projects the person will be working on, a list of groceries to buy, etc.

6. TRANSIT

This section discusses how to describe transit routes for the TRANSIMS planner and microsimulation.

6.1 Terms

Transit Refers to vehicles traveling on pre-specified routes, stopping at specific accessory locations listed in the Transit Stop network data table, and attempting to follow a predetermined schedule.

Route A transit route is a sequential set of transit stops visited by a transit vehicle. Each route is assigned an integer ID. No transit route may include the same transit stop more than once. For example, the *inbound* and *outbound* portions of a round trip must be assigned different route IDs. Also, two transit vehicles that follow the same path through the network but stop at different places along the path (for example, an express and local train) must have different route IDs.

Transit Stop An accessory as defined in the Transit Stop network data table.

6.2 File Format

The transit network topology is described by the *transit route file* (configuration parameter TRANSIT_ROUTE_FILE). This is an ASCII text file listing the Transit Stop IDs at which vehicles on each transit route are allowed to stop.

The transit schedule is described by the *transit schedule file* (configuration parameter TRANSIT_SCHEDULE_FILE). This is a ASCII text file listing information needed by the planner to determine paths through the transit network.

6.2.1 Transit Route File Format

The Transit Route File is an ASCII text file whose fields are separated by white space (space(s), tab(s), or newline(s)). For each route, the file contains the route ID, the number of transit stops the route visits, and a list of the ID of each stop, in the order visited. The column names are not part of the data file.

Table 18: Transit route file data definitions and format.

Column Name	Description	Allowed Values
Transit Route ID	A unique identifier for this route.	integer
Number of Stops	The number of transit stop IDs to follow.	integer
List of Transit Stop IDs	IDs of the transit stops this route visits, in the order encountered.	integer

6.2.2 Transit Schedule File Format

The Transit Schedule File is an ASCII text file whose fields are separated by white space (space(s), tab(s), or newline(s)). The file must be sorted by Vehicle ID, Transit Route ID, and time – in that order.

Table 19: Transit schedule file data definitions and format.

Column Name	Description	Allowed Values
Vehicle ID	Vehicle IDs are defined in the vehicle file.	integer
Transit Route ID	A unique identifier for this route.	integer
Time	Arrival time at the stop.	integer: seconds since midnight
Link ID	IDs of the link on which the transit stop resides.	integers
Destination Node ID	ID of the node toward which the vehicle is heading.	integer
Transit Stop ID	ID of this transit stop, as specified in the network data tables.	integer

6.3 Interface Functions

The transit subsystem has C structures and utility functions that are used to read and write data from a TRANSIMS vehicle file.

The function **`getNextTransit()`** reads transit data from a transit file in ASCII format. The function stores the information in an unmodifiable data structure (`TransitData`), and returns a pointer to the structure. Since the `TransitData` structure cannot be modified by the calling program, the data should be copied if it needs to be changed.

The function **`writeTransit()`** takes a `TransitData` structure as an argument containing the information to be written. The **`getNextTransit()`** function combined with the **`moreTransit()`** function provides a mechanism for iterating through the transit file reading the transit data.

6.3.1 moreTransitData

Signature: `int moreTransitData(FILE * const fp)`

Description: Boolean function used to control iteration through the transit file.

Argument: `fp – FILE *` for the transit file that must be open for reading.

Return Value: Returns 1 if not at end of transit data file.
Returns 0 if EOF has been reached.

6.3.2 getNextTransitData

Signature: `const TransitData * getNextTransitData (FILE * const fp)`

Description: Reads transit data from the transit data file.

Argument: `fp` – `FILE *` to the transit data file, which must be open for reading.

Return Value: The address of a `TransitData` structure containing the transit data read from the file. Returns `NULL` on error.

6.3.3 writeTransitData

Signature: `int writeTransitData(FILE * const fp, TransitData * data);`

Description: Writes the given `TransitData` into a line of the given transit data file.

Argument: `fp` – `FILE *` to the transit data file, which must be open for writing.
`data` – address of a `TransitData` structure containing the data to be written.

Return Value: 1 on success.
0 on error.

6.4 Data Structures

6.4.1 TransitData

This structure is used for transit data as specified in the transit route file.

```
typedef struct transitdata_s
{
/** The route Id. **/
INT32 fRouteId;

/** The number of stops.  **/
INT32 fNumStops;

/** An array of stopIds for the stops. **/
INT32 *fStopIDs;

} TransitData;
```

6.5 Files

Table 20: Transit library files.

Type	File Name	Description
Binary Files	libTIO.a	TRANSIMS Interfaces library
Source Files	transitio.c	Defines transit data structures and interface functions
	transitio.h	Transit interface functions source file

6.6 Configuration Keys

Table 21: Transit file configuration keys.

Configuration Key	Description
TRANSIT_ROUTE_FILE	The name of a transit route file whose format is described Table 18. Used as input by the microsimulation and the Route Planner.
TRANSIT_SCHEDULE_FILE	The name of a transit schedule file whose format is described above. Used as input by the microsimulation.

6.7 Examples

In the example, note that

- routes 1 and 2 together comprise a round trip
- routes 8 and 10 stop in the same places
- routes need not be consecutively numbered
- stops need not be listed in numerical order

```
1 5
5
6
7
8
9
2 6
8
9
6
7
4
5
8 2
92
70
9 2
68
69
10 2
92
70
```

7. NETWORK

The TRANSIMS network representation provides detailed information about streets, intersections, signals, and transit in a road network. This section discusses the concepts involved in describing a road network and the TRANSIMS data table formats. In our analysis of road networks, we have relied on traffic engineering practice as described in references [Do 97], [GHA 88], [ITE 85], [ITE], [MM 84], [Or 93], and [PP 93].

7.1 Terms

Node A node is the part of the network corresponding to a vertex in graph theory. Nodes typically occur at intersections in the road network. A node must be present where the network branches and where the permanent number of lanes changes. A lane is considered permanent if it is not a temporary, pocket lane (see the definition of pocket lane below). A node may be present where neither of the aforementioned occurs, however. Nodes are not required where turn pockets start or end because these are not considered permanent lanes. Each node has a traffic control associated with it (null, unsignalized, pre-timed, actuated, coordinated, etc.).

Link A link is the part of the network corresponding to an edge in graph theory. Links represent street and road segments. Each link has a constant number of permanent lanes but may have a variable number of pocket lanes. A link may have lanes in both directions; alternately, the lanes in opposite directions may be on separate links (in which case no passing into oncoming lanes is possible). Table 22 (at the end of this section) lists the functional classes for links.

Lane A lane is where traffic flows on a link. The lanes on each side/direction of the link are numbered separately, starting with lane number one as the leftmost lane (relative to the direction of travel). Each successive lane to the right of it is numbered one greater than its predecessor. Pocket lanes (i.e., turn pockets, merges, and pull-outs) are numbered in sequence, even if they do not exist for the full length of the link. A two-way left-turn lane, if present, is considered to be lane number zero.

Pocket Lanes A pocket lane is either (a) a right- or left-turn pocket (a lane that starts after the *from* node and ends at the *to* node), (b) a right or left pull-out (a lane that starts after the *from* node and ends before the *to* node), or (c) a right or left merge pocket (a lane that starts at the *from* node and ends before the *to* node). If a lane starts at the *from* node and ends at the *to* node, it is considered a permanent lane, not a pocket lane.

Barrier A barrier is a divider such as a curb or grade separation that prevents vehicles from moving between two adjacent lanes on a link.

Parking Parking areas are located along links and are used as origins and destinations for vehicle trips. Parking may be placed where it is physically

located in the network, or it may be placed in aggregate generic parking areas representing several of the driveways, lots, parking places, etc., on a link. Places where vehicles leave the network are called boundary parking areas.

Transit Stop A transit stop is a location on a link where a transit vehicle, such as a bus or light rail car, waits to embark and disembark passengers.

Lane Connectivity Lane connectivity specifies how lanes are connected across a node. Lanes are numbered from the median and include turn pockets. Incoming and outgoing links and lanes are defined relative to the node. For each incoming lane on an incoming link, at least one outgoing lane must be specified for each outgoing link that a vehicle on the incoming link can transition to. Multiple outgoing lanes may be defined for an outgoing link, if desired.

Traffic Control Each node has a traffic control associated with it. The traffic control specifies how lanes are connected across the node and the type of sign or signalized control that determines who has the right-of-way.

Signal Coordinator A signal coordinator is a device that controls the operation of one or more traffic controls.

Unsignalized Node An unsignalized node represents the type of sign control, if any, that is present at an unsignalized node. Examples are stop and yield signs. Nodes where only the number of permanent lanes is changing are generally considered unsignalized.

Signalized Node A signalized node represents a traffic light. Each signal has a timing plan and a phasing plan.

Phasing Plan A phasing plan specifies the turn protection in effect for transitioning from an incoming link to an outgoing link during a particular phase of a specific timing plan.

Timing Plan A timing plan specifies the lengths of the intervals during the specific phases for a traffic light. Many nodes may have the same timing plan. It is possible for each phase to transition to more than one phase if required.

Detector A detector is a device that identifies the presence or passage of a vehicle over an area of the lanes on a link.

Activity Location An activity location is a place on a link where traveler activities (such as work, home, shopping) can take place.

Process Link A process link is a “virtual” connection between an activity location, parking place, or transit stop and another activity location, parking location, or transit stop; it represents the process of changing modes and accounts for the cost (in time and money) of making a mode change.

Study/Buffer Areas The microsimulation distinguishes two types of links in its calculations: Study area links are the links of interest for the traffic analyst. The output

subsystem, for instance, records events such as when a vehicle leaves or enters the study area. The nature of the microsimulation makes it necessary to simulate traffic on additional buffer area links. Typically, these links form a fringe about two links thick around the study area. A simulation includes buffer links in order to avoid edge effects such as when vehicles enter the study area on its boundary; the buffer gives these vehicles time to interact with other traffic and achieve realistic behavior before entering the study area.

Table 22: Functional classes for links [Do 97].

Name	Interpretation
Freeway	A divided, arterial highway for through traffic with full control of access. Full access control means the authority to control access is exercised to give preference to through traffic by providing access connections with selected public roads, but prohibiting grade crossings and/or direct private driveway connections.
Expressway	A divided, arterial highway for through traffic with partial control of access. Partial control of access means that some authority is exercised to control access in the manner described above, but there are crossings at grade and/or direct private driveway connections.
Primary Arterial	A major arterial roadway with intersections at grade crossings and direct access to abutting property and on which geometric design and traffic-control measures are used to expedite safe movement of through traffic.
Secondary Arterial	A minor arterial roadway with intersections at grade crossings and direct access to abutting property and on which geometric design and traffic-control measures are used to expedite safe movement of through traffic.
Frontage Road	An arterial that runs parallel to a freeway or expressway.
Collector Street	A roadway on which vehicular traffic is given preferential right of way, and at the entrances to which vehicular traffic from intersecting roadways is required by law to yield right-of-way to vehicles on such a roadway in obedience to either a stop sign or a yield sign, when such signs are erected.
Local Street	A street or road primarily for access to residence, business, or other abutting property.
Freeway Ramp	A unidirectional roadway providing connection between a freeway or expressway and an arterial.
Zonal Connector	An imaginary (non-physical) connection to or from the centroid of a traffic analysis zone.
Other	Any roadway not fitting the above definitions.
Walkway	A street restricted to use by pedestrians.
Busway	A street restricted to use by buses.
Light Rail	A roadbed restricted to use by light rail cars.
Heavy Rail	A roadbed restricted to use by heavy rail cars.
Ferry	A waterway crossed by ferry.

7.2 File Format

This section specifies the formats for the 19 data tables required to describe a TRANSIMS road network. Table 23 shows how the tables depend on one another. The units of measurement are SI units—i.e., distances in meters, time in seconds, etc. Geographic coordinates are specified in the UTM system. The TRANSIMS software architecture allows for the inclusion of additional columns desired by an analyst, so the specification below gives only the required columns. The format for data files is ASCII, with columns delimited by tab characters; records are terminated by a new-line character (i.e., ISO format). The first line of the file must contain the field names (i.e., column headings) delimited by tab characters.

Table 23: Interdependencies between network tables.

Table	Tables on which it depends
Link	Node
Speed	Node, Link, Pocket Lane
Pocket Lane	Node, Link
Lane Use	Node, Link, Pocket Lane
Parking	Node, Link
Barrier	Node, Link, Pocket Lane
Transit Stop	Node, Link
Lane Connectivity	Node, Link, Pocket Lane
Turn Prohibition	Node, Link, Pocket Lane
Unsignalized Node	Node, Link, Pocket Lane
Signalized Node	Node, Timing Plan
Phasing Plan	Node, Link, Pocket Lane, Timing Plan
Detector	Node, Link, Pocket Lane
Signal Coordinator	Node, Signalized Node
Activity Location	Node, Link
Process Link	Parking, Transit Stop, Activity Location
Study Area Link	Link

7.2.1 Node Table

Table 24 specifies the format for the node table. To validate a node table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.
- The IDs are unique.
- No nodes have the same easting, northing, and elevation. Nodes with the same easting and northing, but different elevations, are acceptable.

Table 24: Node table format.

Column Name	Description	Allowed Values
ID	ID number of the node.	integer: 1 through 2,147,483,647
EASTING	The *x*-coordinate of the node (in meters, UTM coordinate system).	floating-point number
NORTHING	The *y*-coordinate of the node (in meters, UTM coordinate system).	floating-point number
ELEVATION	The *z*-coordinate of the node (in meters, UTM coordinate system).	floating-point number
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

7.2.2 Link Table

Table 25 specifies the format for the link table. To validate a link table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.
- The IDs are unique.
- The nodes at the endpoints exist.
- There are different nodes at the endpoints.
- There are permanent lanes in at least one direction.
- There is at least one permanent lane in every direction that there is a pocket lane.
- The length of the link is at least as great as the distance between its endpoints.
- The length of the link is not far greater (e.g., 50% more) than the distance between its endpoints.
- The length of the link is not exceedingly small. (The TRANSIMS microsimulation may have difficulty simulating successive links that are less than about 50 meters long.)
- The sum of the setback lengths is less than the length of the link.
- All nodes have at least one incoming and one outgoing link.
- At least some types of vehicles are allowed on the link.
- The functional classes of all of the links connected to a node are consistent: Divide the TRANSIMS functional classes into three categories: (i) restricted—Freeway, Expressway; (ii) surface—Primary Arterial, Secondary Arterial, Frontage Road, Collector, Local Street, Zonal Connector, Other, Ferry, Walkway; (iii) miscellaneous—Ramp, Bikeway, Busway, Light Rail, Heavy Rail. There are inconsistent functional classes if there is a mixture of *restricted* and *surface* links at a node. (This notion can probably be refined further.)
- The network graph is fully connected (i.e., one can reach any node from any other node).
- The network does not contain modal *sources* or *sinks*. A modal source (sink) is a node vehicles of a particular type can leave (enter), but cannot enter (leave).
- The network does not contain unwanted modal *islands*. A modal island is a set of links for a particular type of vehicle that is disconnected from the rest of the links for that type of vehicle. (There may be some cases, such as for transit routes, where modal islands are desirable.)

Table 25: Link table format.

Column Name	Description	Allowed Values
ID	ID number of the link.	integer: 1 through 2,147,483,647
NAME	Name of street.	50 characters
NODEA	ID number of the node at A.	integer: 1 through 2,147,483,647
NODEB	ID number of the node at B.	integer: 1 through 2,147,483,647
PERMLANESA	Number of lanes on the link heading toward node A, not including pocket lanes.	integer: 0 through 255
PERMLANESB	Number of lanes on the link heading toward node B, not including pocket lanes.	integer: 0 through 255
LEFTPCKTSA	Number of pocket lanes to the left of the permanent lanes heading toward node A.	integer: 0 through 255
LEFTPCKTSB	Number of pocket lanes to the left of the permanent lanes heading toward node B.	integer: 0 through 255
RGHTPCKTSA	Number of pocket lanes to the right of the permanent lanes heading toward node A.	integer: 0 through 255
RGHTPCKTSB	Number of pocket lanes to the right of the permanent lanes heading toward node B.	integer: 0 through 255
TWOWAYTURN	Whether there is a two-way left-turn lane in the center of the link.	one character: 'F' = false/no 'T' = true/yes
LENGTH	Length of the link (in meters).	positive floating-point number
GRADE	Percentage grade from node A to node B, uphill being a positive number.	floating-point number between –100 and +100
SETBACKA	Set-back distance (in meters) from the center of the intersection at node A.	non-negative floating-point number
SETBACKB	Set-back distance (in meters) from the center of the intersection at node B.	non-negative floating-point number
CAPACITYA	Total capacity (in vehicles per hour) for the lanes traveling to node A.	non-negative floating-point number
CAPACITYB	Total capacity (in vehicles per hour) for the lanes traveling to node B.	non-negative floating-point number
SPEEDLMTA	Default speed limit (in meters per second) for vehicles traveling toward node A.	positive floating-point number
SPEEDLMTB	Default speed limit (in meters per second) for vehicles traveling toward node B.	positive floating-point number
FREESPDA	Default free-flow speed (in meters per second) for vehicles traveling toward node A.	positive floating-point number
FREESPDB	Default free-flow speed (in meters per second) for vehicles traveling toward node B.	positive floating-point number

Column Name	Description	Allowed Values
FUNCTCLASS	Functional class of the link; a link that permits both road and rail traffic should be coded with the roadway class.	ten characters: ‘FREEWAY’ = freeway ‘XPRESSWAY’ = expressway ‘PRIARTER’ = primary arterial ‘SECARTER’ = secondary arterial ‘FRONTAGE’ = frontage road ‘COLLECTOR’ = collector ‘LOCAL’ = local street ‘RAMP’ = freeway ramp ‘ZONECONN’ = zonal connector ‘OTHER’ = other ‘WALKWAY’ = walk only ‘BIKEWAY’ = bicycle only ‘BUSWAY’ = bus only roadway ‘LIGHTRAIL’ = light rail only ‘HEAVYRAIL’ = heavy rail ‘FERRY’ = ferry
THRUA	Default through link connected at node A; a zero indicates there is no through link.	integer: 0 through 2,147,483,647
THRUB	Default through link connected at node B; a zero indicates there is no through link.	integer: 0 through 2,147,483,647
COLOR	The color number for the link (all of the links connected to a given link must have different colors).	integer: 1 through 63
VEHICLE	Vehicle types (modes) allowed to use this link.	string of characters separated by slashes: ‘WALK’ = walking allowed ‘AUTO’ = private auto ‘TRUCK’ = motor carrier ‘BICYCLE’ = bicycle ‘TAXI’ = paratransit ‘BUS’ = bus ‘TROLLEY’ = trolley ‘STREETCAR’ = streetcar ‘LIGHTRAIL’ = light rail transit ‘RAPIDRAIL’ = rail rapid transit ‘REGIONRAIL’ = regional rail
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

7.2.3 Speed Table

Entries in the Speed Table are only required when the speed limit or free speed for a link varies for different types of vehicles allowed to use the link. The speeds that appear in the Link Tables are used as defaults for any vehicle types not specified in a record in the Speed Table.

Table 26 specifies the format for the speed table. To validate a speed table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.

- The node and link references are correct.
- The vehicle types are consistent with the vehicle types allowed on the link.

Table 26: Speed table format.

Column Name	Description	Allowed Values
LINK	ID number of the link with multiple speeds.	integer: 1 through 2,147,483,647
NODE	ID number of the node toward which lanes are headed.	integer: 1 through 2,147,483,647
SPEEDLMT	Speed limit (in meters per second) for vehicles.	positive floating-point number
FREESPD	Free-flow speed (in meters per second) for vehicles.	positive floating-point number
VEHICLE	Vehicle type(s) to which speeds apply.	string of characters separated by slashes: 'AUTO' = private auto 'TRUCK' = motor carrier 'BICYCLE' = bicycle 'TAXI' = paratransit 'BUS' = bus 'TROLLEY' = trolley 'STREETCAR' = streetcar 'LIGHTRAIL' = light rail transit 'RAPIDRAIL' = rail rapid transit 'REGIONRAIL' = regional rail
STARTTIME	Starting time for the speeds.	a character string with the day of week, 'SUN' = Sunday 'MON' = Monday 'TUE' = Tuesday 'WED' = Wednesday 'THU' = Thursday 'FRI' = Friday 'SAT' = Saturday 'WKE' = any weekend day 'WKD' = any weekday 'ALL' = any day, followed by the time of day (on a 24-hour clock), for example 'WKD13:20' is any weekday at 1:20 in the afternoon
ENDTIME	Ending time for the speeds.	specified like STARTTIME
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

7.2.4 Pocket Lane Table

Table 27 specifies the format for the pocket lane table. To validate a pocket lane table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.
- The IDs are unique.
- The node and link references are correct.
- The lane number is that of a valid pocket lane.
- The offset and length are consistent with the setbacks and length of the link.
- None of the pockets overlap.
- All of the pocket lanes specified in the link table are present.

Table 27: Pocket lane table format.

Column Name	Description	Allowed Values
ID	ID number of the pocket lane.	integer: 1 through 2,147,483,647
NODE	ID number of the node toward which the pocket lane leads.	integer: 1 through 2,147,483,647
LINK	ID number of the link on which the pocket lane lies.	integer: 1 through 2,147,483,647
OFFSET	Starting position of the pocket lane, measured (in meters) from NODE (applicable to pull-out pockets only).	non-negative floating-point number
LANE	Lane number of the pocket lane.	integer: 1 through 255
STYLE	Type of the pocket lane.	one character: 'T' = turn pocket 'P' = pull-out pocket 'M' = merge pocket
LENGTH	Length of the pocket lane (in meters); turn pockets and merge pockets always start or end at the appropriate limit line.	positive floating-point number
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

7.2.5 Lane Use Table

Entries in the Lane Use Table are only required when a lane has restrictions for certain vehicle types. The vehicle types specified in the Link Table are permitted unrestricted use of all lanes on the link when there is no record in the Lane Use Table.

Table 28 specifies the format for the lane use table. To validate a lane use table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.
- The node, link, and lane references are correct.
- The vehicle types allowed for the parking are consistent with the vehicle types allowed on the link.

Table 28: Lane use table format.

Column Name	Description	Allowed Values
NODE	ID number of the node toward which the lane leads.	integer: 1 through 2,147,483,647
LINK	ID number of the link on which the lane lies.	integer: 1 through 2,147,483,647
LANE	Lane number.	integer: 1 through 255
VEHICLE	Vehicle type(s) to which restriction applies.	string of characters separated by slashes: 'HOV2' = high occupancy vehicle with two or more occupants 'HOV3' = high occupancy vehicle with three or more occupants 'HOV4' = high occupancy vehicle with four or more occupants 'BICYCLE' = bicycle 'AUTO' = private auto 'TRUCK' = motor carrier 'BUS' = bus 'TROLLEY' = trolley 'STREETCAR' = streetcar 'LIGHTRAIL' = light rail transit 'RAPIDRAIL' = rail rapid transit 'REGIONRAIL' = regional rail
RESTRICT	Type of lane restriction.	one character: 'O' = only this vehicle type may use lane 'R' = lane required to be used by this vehicle type 'N' = lane not allowed to be used by this vehicle type
STARTTIME	Starting time for the restriction.	a character string with the day of week, 'SUN' = Sunday 'MON' = Monday 'TUE' = Tuesday 'WED' = Wednesday 'THU' = Thursday 'FRI' = Friday 'SAT' = Saturday 'WKE' = any weekend day 'WKD' = any weekday 'ALL' = any day, followed by the time of day (on a 24-hour clock), for example 'WKD13:20' is any weekday at 1:20 in the afternoon
ENDTIME	Ending time for the restriction.	specified like STARTTIME
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

7.2.6 Parking Table

Table 29 specifies the format for the parking table. To validate a parking table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.
- The IDs are unique.
- The node and link references are correct.
- The offset is consistent with the setbacks and length of the link.
- The vehicle types allowed for the parking are consistent with the vehicle types allowed on the link.

Table 29: Parking table format.

Column Name	Description	Allowed Values
ID	ID number of the parking place.	integer: 1 through 2,147,483,647
NODE	ID number of the node toward which vehicles are traveling.	integer: 1 through 2,147,483,647
LINK	ID number of the link on which the parking place lies.	integer: 1 through 2,147,483,647
OFFSET	Location of the entrance from the link to the parking place, measured (in meters) from NODE.	non-negative floating-point number
STYLE	Type of the parking place.	five characters: 'PRSTR' = parallel on street 'HISTR' = head in on street 'DRVWY' = driveway 'LOT' = parking lot 'BNDRY' = network boundary
CAPACITY	Number of vehicles the parking place can accommodate; zero for unlimited capacity.	integer: 0 through 65,535
GENERIC	Whether the parking place represents generic parking (not an actual driveway, lot, etc., but a group/aggregate of them used to simplify modeling).	one character: 'T' = true/yes 'F' = false/no
VEHICLE	Type of vehicle(s) allowed to park at the parking place.	string of characters separated by slashes: 'AUTO' = private auto 'TRUCK' = motor carrier 'BICYCLE' = bicycle 'TAXI' = paratransit 'BUS' = bus 'TROLLEY' = trolley 'STREETCAR' = streetcar 'LIGHTRAIL' = light rail transit 'RAPIDRAIL' = rail rapid transit 'REGIONRAIL' = regional rail 'ANY' = any vehicle type
STARTTIME	Starting time for parking.	a character string with the day of week, 'SUN' = Sunday 'MON' = Monday 'TUE' = Tuesday 'WED' = Wednesday 'THU' = Thursday 'FRI' = Friday 'SAT' = Saturday 'WKE' = any weekend day 'WKD' = any weekday 'ALL' = any day, followed by the time of day (on a 24-hour clock), for example 'WKD13:20' is any weekday at 1:20 in the afternoon
ENDTIME	Ending time for parking.	specified like STARTTIME
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

7.2.7 Barrier Table

☛ Barriers are not supported in IOC-2.

Table 30 specifies the format for the barrier table. To validate a barrier table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.
- The IDs are unique.
- The node, link, and lane references are correct.
- The offset and length are consistent with the setbacks and length of the link.

Table 30: Barrier table format.

Column Name	Description	Allowed Values
ID	ID number of the barrier.	integer: 1 through 2,147,483,647
NODE	ID number of the node toward which vehicles are traveling.	integer: 1 through 2,147,483,647
LINK	ID number of the link on which the barrier lies.	integer: 1 through 2,147,483,647
OFFSET	Starting position of the barrier, measured (in meters) from NODE.	non-negative floating-point number
LANE	Lane number of lane to the left of the barrier.	integer: 0 through 255
STYLE	Type of the barrier.	ten characters: ‘CURB’ = curb ‘BARRIER’ = barrier ‘GRADESEP’ = grade separation ‘STRIPE’ = painted stripe ‘TEMPORARY’ = temporary barrier
LENGTH	Length of the barrier (in meters).	positive floating-point number
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

7.2.8 Transit Stop Table

Table 31 specifies the format for the transit stop table. To validate a transit stop table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.
- The IDs are unique.
- The node and link references are correct.
- The offset is consistent with the setbacks and length of the link.
- The vehicle types allowed for the transit stop are consistent with the vehicle types allowed on the link.

Table 31: Transit stop table format.

Column Name	Description	Allowed Values
ID	ID number of the stop.	integer: 1 through 2,147,483,647
NAME	Name of the stop.	50 characters
NODE	ID number of the node toward which vehicles are traveling.	integer: 1 through 2,147,483,647
LINK	ID number of the link on which the stop occurs.	integer: 1 through 2,147,483,647
OFFSET	Location of the stop, measured (in meters) from NODE.	non-negative floating-point number
VEHICLE	Types of vehicles for which this is a stop.	string of characters separated by slashes: 'BUS' = bus 'TROLLEY' = trolley 'STREETCAR' = streetcar 'LIGHTRAIL' = light rail transit 'RAPIDRAIL' = rail rapid transit 'REGIONRAIL' = regional rail
STYLE	Type of the stop.	ten characters: 'STOP' = stop (no station) 'STATION' = station 'YARD' = vehicle storage lot
CAPACITY	Number of vehicles the stop can simultaneously handle; zero for unlimited capacity.	integer: 0 through 65,535
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

7.2.9 Lane Connectivity Table

Table 32 specifies the format for the lane connectivity table. To validate a lane connectivity table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.
- The node, link, and lane references are correct.
- Each lane has at least one incoming and at least one outgoing connection.

Table 32: Lane Connectivity table format.

Column Name	Description	Allowed Values
NODE	ID number of the node.	integer: 1 through 2,147,483,647
INLINK	ID number of the incoming link.	integer: 1 through 2,147,483,647
INLANE	Lane number of the incoming lane.	integer: 1 through 255
OUTLINK	ID number of the outgoing link.	integer: 1 through 2,147,483,647
OUTLANE	Lane number of the outgoing lane.	integer: 1 through 255
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

7.2.10 Turn Prohibition Table

Entries in the Turn Prohibition Table are required when particular movements at a node are prohibited only at certain times of the day. The Lane Connectivity Table specifies the allowed and prohibited movements that are always in effect at a node.

☛ A Turn Prohibition Table is not required in IOC-2 because time-of-day restrictions are not currently supported.

Table 33 specifies the format for the turn prohibition table. To validate a turn prohibition table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.
- The node and link references are correct.

Table 33: Turn prohibition table format.

Column Name	Description	Allowed Values
NODE	ID number of the node.	integer: 1 through 2,147,483,647
INLINK	ID number of the incoming link.	integer: 1 through 2,147,483,647
OUTLINK	ID number of the outgoing link.	integer: 1 through 2,147,483,647
STARTTIME	Starting time for the prohibition.	a character string with the day of week, 'SUN' = Sunday 'MON' = Monday 'TUE' = Tuesday 'WED' = Wednesday 'THU' = Thursday 'FRI' = Friday 'SAT' = Saturday 'WKE' = any weekend day 'WKD' = any weekday 'ALL' = any day, followed by the time of day (on a 24-hour clock), for example 'WKD13:20' is any weekday at 1:20 in the afternoon
ENDTIME	Ending time for the prohibition.	specified like STARTTIME
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

7.2.11 Unsignalized Node Table

Table 34 specifies the format for the unsignalized node table. To validate an unsignalized node table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.
- The node and link references are correct.
- Each incoming link entering an unsignalized node has a record.

Table 34: Unsignalized node table format.

Column Name	Description	Allowed Values
NODE	ID number of the node.	integer: 1 through 2,147,483,647
INLINK	ID number of the incoming link.	integer: 1 through 2,147,483,647
SIGN	Type of sign control on the link.	one character: 'S' = stop 'Y' = yield 'N' = none
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

7.2.12 Signalized Node Table

Table 35 specifies the format for the signalized node table. To validate a signalized node table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.
- The node references are correct.
- The plan references are correct.
- Each node has either one signalized or one unsignalized control.
- All plans are used.
- The start times are valid.

Table 35: Signalized node table format.

Column Name	Description	Allowed Values
NODE	ID number of the node.	integer: 1 through 2,147,483,647
TYPE	Type of the signal.	one character: 'T' = timed 'A' = actuated
PLAN	ID number of a timing plan.	integer: 1 through 65,535
OFFSET	Relative offset (in seconds) for coordinated signals.	non-negative floating-point number
STARTTIME	Starting time for the plan.	a character string with the day of week 'SUN' = Sunday 'MON' = Monday 'TUE' = Tuesday 'WED' = Wednesday 'THU' = Thursday 'FRI' = Friday 'SAT' = Saturday 'WKE' = any weekend day 'WKD' = any weekday 'ALL' = any day, followed by the time of day (on a 24-hour clock), for example 'WKD13:20' is any weekday at 1:20 in the afternoon
COORDINATR	ID number of coordinator for the signal; equivalent to NODE number if signal is isolated.	integer: 1 through 2,147,483,647
RING	Single or dual ring, required only for TYPE = 'A'.	one character: 'S' = single 'D' = dual
ENTRY	Single or dual entry, required only for RING = 'D'.	one character: 'S' = single 'D' = dual
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

7.2.13 Phasing Plan Table

Table 36 specifies the format for the phasing plan table. To validate a phasing plan table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.
- The plan, phase, node, and link references are correct.
- Each incoming and outgoing link is controlled.

Table 36: Phasing plan table format.

Column Name	Description	Allowed Values
NODE	ID number of the node .	integer: 1 through 2,147,483,647
PLAN	ID number of the timing plan.	integer: 1 through 65,535
PHASE	Phase number.	integer: 1 through 255
INLINK	ID number of the incoming link.	integer: 1 through 2,147,483,647
OUTLINK	ID number of the outgoing link.	integer: 1 through 2,147,483,647
PROTECTION	Movement protection indicator.	one character: 'P' = protected 'U' = unprotected 'S' = unprotected after stop
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

7.2.14 Timing Plan Table

Table 37 specifies the format for the timing plan table. To validate a timing plan table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.
- The (plan, phase) pairs are unique.
- The time values are consistent.
- The phase sequence references existent phases.

Table 37: Timing plan table format.

Column Name	Description	Allowed Values
PLAN	ID number of a timing plan.	integer: 1 through 65,535
PHASE	Phase number.	integer: 1 through 255
NEXTPHASES	Phase number(s) of the next phase(s) in sequence.	string of phase numbers, separated by slashes
GREENMIN	Minimum length (in seconds) of the green interval, or fixed green length for timed signal.	non-negative floating-point number
GREENMAX	Maximum length (in seconds) of the green interval.	non-negative floating-point number
GREENEXT	Length (in seconds) of the green extension interval.	non-negative floating-point number
YELLOW	Length (in seconds) of the yellow interval.	non-negative floating-point number
REDCLEAR	Length (in seconds) of the red clearance interval.	non-negative floating-point number
GROUPFIRST	For pre-timed or single ring: 1 if first phase, 0 if not first phase; for dual ring: number of phase group for which this phase is first phase, 0 if not first phase in the phase group.	integer: 0 through 255
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

7.2.15 Detector Table

☛ Detectors are not supported in IOC-2.

Table 38 specifies the format for the detector table. To validate a detector table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.
- The IDs are unique.
- The node, link, and lane references are correct.
- The offset and length are consistent with the setbacks and length of the link.

Table 38: Detector table format.

Column Name	Description	Allowed Values
ID	ID number of the detector.	integer: 1 through 2,147,483,647
NODE	ID number of the node toward which vehicles are traveling.	integer: 1 through 2,147,483,647
LINK	ID number of the link on which the detector lies.	integer: 1 through 2,147,483,647
OFFSET	Starting position of the detector, measured (in meters) from NODE.	non-negative floating-point number
LANEBEGIN	Lane number of lane at which the detector begins.	integer: 1 through 255
LANEEND	Lane number of lane at which the detector ends, equal to LANEBEGIN for detector that lies on single lane.	integer: 1 through 255
LENGTH	Length of the detector (in meters).	non-negative floating-point number
STYLE	Type of the detector.	ten characters: 'PRESENCE' = sense vehicles on detector 'PASSAGE' = sense vehicles crossing detector
COORDINATR	ID number of coordinators interested in detector output.	string of coordinator IDs separated by slashes
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

7.2.16 Signal Coordinator Table

☛ Signal Coordinators are not supported in IOC-2.

Table 39 specifies the format for the signal coordinator table. To validate a signal coordinator table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.
- The IDs are unique.

Table 39: Signal coordinator table format.

Column Name	Description	Allowed Values
ID	ID number of the signal coordinator.	integer: 1 through 2,147,483,647
TYPE	Type of coordinator.	ten characters: values to be determined
ALGORITHM	Control algorithm used by coordinator.	ten characters: values to be determined
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

7.2.17 Activity Locations

Table 40 specifies the format for the activity location table. To validate an activity location table, it is necessary to verify the following.

- The field names and types are correct.
- The data values are in the legal ranges.
- The IDs are unique.
- The node and link references are correct.
- The offset is consistent with the setbacks and lengths of the links.
- The layer is consistent with the vehicle types allowed on the link.
- The names of any optional user-defined fields are unique within the table.

Table 40: Activity locations.

Column Name	Description	Allowed Values
ID	ID number of the activity location.	integer: 1 through 2,147,483,647
NODE	ID number of the node toward which vehicles are traveling (the location is taken to be on the right side of the street when headed this direction).	integer: 1 through 2,147,483,647
LINK	ID number of the link on which the activity location lies.	integer: 1 through 2,147,483,647
OFFSET	Location of the entrance from the link to the activity location, measured (in meters) from NODE.	non-negative floating-point number
LAYER	The modal "layer" on which the activity location resides.	string of characters: "AUTO" or "BUS" or "LIGHTRAIL" or "WALK"
EASTING	The *x*-coordinate of the node (in meters, UTM coordinate system).	floating-point number
NORTHING	The *y*-coordinate of the node (in meters, UTM coordinate system).	floating-point number
ELEVATION	The *z*-coordinate of the node (in meters, UTM coordinate system).	floating-point number
optional field 1	First optional field related to land use.	floating-point number
optional field 2	Second optional field related to land use.	floating-point number
optional field n	The *n*-th optional field related to land use.	floating point number
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

A maximum of 20 user-defined fields may optionally be included in the table between the ELEVATION and NOTES fields. These optional fields are typically related to land use, but could be anything the user wishes to specify about an activity location. The column names may be up to 32 characters in length. The presence of any optional fields is detected by the **`NetReadActivityLocationHeader()`** function. This implies that the header for the activity

location table must be read by this function rather than by **`NetReadHeader()`** or **`NetSkipHeader()`**, whether or not optional fields are included.

7.2.18 Process Links

Table 41 specifies the format for a process link table. To validate a process link table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.
- The IDs are unique.
- The "from" and "to" accessory references are correct.

Table 41: Process links.

Column Name	Description	Allowed Values
ID	ID number of the virtual link.	integer: 1 through 2,147,483,647
FROMID	ID number of the accessory from which the virtual link leaves.	integer: 1 through 2,147,483,647
FROMTYPE	Type of accessory from which the virtual link leaves.	string of characters: "ACTIVITY" or "PARKING" or "TRANSIT"
TOID	ID number of the accessory to which the virtual link leads.	integer: 1 through 2,147,483,647
TOTYPE	Type of accessory to which the virtual link leads.	string of characters: "ACTIVITY" or "PARKING" or "TRANSIT"
DELAY	The time delay (measured in seconds) incurred when traveling across the virtual link.	non-negative floating-point number
COST	The cost (measured in arbitrary units) incurred when traveling across the virtual link.	non-negative floating-point number
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

Note that although the costs are measured in arbitrary units, the units must be the same for the whole data table.

7.2.19 Study Area Link Table

Table 42 specifies the format for the study area link table. To validate a study area link table, it is necessary to verify the following:

- The field names and types are correct.
- The data values are in the legal ranges.
- The link references are correct.

Table 42: Study area link table format.

Column Name	Description	Allowed Values
ID	ID number of the link.	integer: 1 through 2,147,483,647
BUFFER	Whether the link is in the buffer area or the study area.	one character: 'Y' = in buffer area 'N' = in study area
NOTES	Character string used for data quality annotations; free format (may be blank).	255 characters

7.3 Interface Functions

The network subsystem has C structures and utility functions for reading and writing network data files.

7.3.1 NetReadHeader

Signature: `int NetReadHeader(FILE * file, TNetHeader * header)`

Description: Read a header from a network table.

Argument: `file` – `FILE` pointer for the network data table.
`header` – pointer to `TNetHeader` structure into which the header is read.

Return Value: Return nonzero if the header was successfully read, or zero if not.

7.3.2 NetWriteHeader

Signature: `int NetWriteHeader(FILE * file, const TNetHeader * header)`

Description: Write a header from a network table.

Argument: `file` – `FILE` pointer for the network data table.
`header` – pointer to `TNetHeader` structure from which the header is written.

Return Value: Return nonzero if the header was successfully written, or zero if not.

7.3.3 NetSkipHeader

Signature: `int NetSkipHeader(FILE * file)`

Description: Skip a header from a network table.

Argument: `file` – `FILE` pointer for the network data table.

Return Value: Return nonzero if the header was successfully skipped, or zero if not.

7.3.4 NetReadActivityLocationHeader

Signature: `int NetReadActivityLocationHeader(FILE* file, TNetHeader* header, TNetActivityLocationRecord* record)`

Description: Read a header from an activity location table.

Argument: `file` – `FILE` pointer for the network data table.
`header` – pointer to `TNetHeader` structure into which the header is read.
`record` – pointer to `TNetActivityLocationRecord` structure which is initialized based on the header contents.

Return Value: Return nonzero if the header was successfully read, or zero if not.

7.3.5 NetReadNode

Signature: `int NetReadNode(FILE * file, TNetNodeRecord * record)`

Description: Read a record from a node table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetNodeRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.6 NetWriteNode

Signature: `int NetWriteNode(FILE * file, const TNetNodeRecord * record)`

Description: Write a record to a node table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetNodeRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.7 NetReadLink

Signature: `int NetReadLink(FILE * file, TNetLinkRecord * record)`

Description: Read a record from a link table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetLinkRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.8 NetWriteLink

Signature: `int NetWriteLink(FILE * file, const TNetLinkRecord * record)`

Description: Write a record to a link table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetLinkRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.9 NetReadSpeed

Signature: `int NetReadSpeed(FILE * file, TNetSpeedRecord * record)`

Description: Read a record from a speed table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetSpeedRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.10 NetWriteSpeed

Signature: `int NetWriteSpeed(FILE * file, const TNetSpeedRecord * record)`

Description: Write a record to a speed table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetSpeedRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.11 NetReadPocket

Signature: `int NetReadPocket(FILE * file, TNetPocketRecord * record)`

Description: Read a record from a pocket lane table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetPocketRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.12 NetWritePocket

Signature: `int NetWritePocket(FILE * file, const TNetPocketRecord * record)`

Description: Write a record to a pocket lane table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetPocketRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.13 NetReadLaneUse

Signature: `int NetReadLaneUse(FILE * file, TNetLaneUseRecord * record)`

Description: Read a record from a lane use table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetLaneUseRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.14 NetWriteLaneUse

Signature: `int NetWriteLaneUse(FILE * file, const TNetLaneUseRecord * record)`

Description: Write a record to a lane use table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetLaneUseRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.15 NetReadParking

Signature: `int NetReadParking(FILE * file, TNetParkingRecord * record)`

Description: Read a record from a parking table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetParkingRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.16 NetWriteParking

Signature: `int NetWriteParking(FILE * file, const TNetParkingRecord * record)`

Description: Write a record to a parking table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetParkingRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.17 NetReadBarrier

Signature: `int NetReadBarrier(FILE * file, TNetBarrierRecord * record)`

Description: Read a record from a barrier table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetBarrierRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.18 NetWriteBarrier

Signature: `int NetWriteBarrier(FILE * file, const TNetBarrierRecord * record)`

Description: Write a record to a barrier table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetBarrierRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.19 NetReadTransitStop

Signature: `int NetReadTransitStop(FILE * file, TNetTransitStopRecord* record)`

Description: Read a record from a transit stop table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetTransitStopRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.20 NetWriteTransitStop

Signature: `int NetWriteTransitStop(FILE * file, const TNetTransitStopRecord * record)`

Description: Write a record to a transit stop table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetTransitStopRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.21 NetReadLaneConnectivity

Signature: `int NetReadLaneConnectivity(FILE * file, TNetLaneConnectivityRecord * record)`

Description: Read a record from a lane connectivity table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetLaneConnectivityRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.22 NetWriteLaneConnectivity

Signature: `int NetWriteLaneConnectivity(FILE * file, const TNetLaneConnectivityRecord * record)`

Description: Write a record to a lane connectivity table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetLaneConnectivityRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.23 NetReadTurnProhibition

Signature: `int NetReadTurnProhibition(FILE * file, TNetTurnProhibitionRecord * record)`

Description: Read a record from a turn prohibition table.

Argument: `file` – `FILE` pointer for the network data table
`record` – pointer to `TNetTurnProhibitionRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.24 NetWriteTurnProhibition

Signature: `int NetWriteTurnProhibition(FILE * file, const TNetTurnProhibitionRecord * record)`

Description: Write a record to a turn prohibition table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetTurnProhibitionRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.25 NetReadUnsignalizedNode

Signature: `int NetReadUnsignalizedNode(FILE * file, TNetUnsignalizedNodeRecord * record)`

Description: Read a record from an unsignalized node table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetUnsignalizedNodeRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.26 NetWriteUnsignalizedNode

Signature: `int NetWriteSignalizedNode(FILE * file, const TNetUnsignalizedNodeRecord * record)`

Description: Write a record to an unsignalized node table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetUnsignalizedNodeRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.27 NetReadSignalizedNode

Signature: `int NetReadSignalizedNode(FILE * file, TNetSignalizedNodeRecord * record)`

Description: Read a record from a signalized node table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetSignalizedNodeRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.28 NetWriteSignalizedNode

Signature: `int NetWriteSignalizedNode(FILE * file, const TNetSignalizedNodeRecord * record)`

Description: Write a record to a signalized node table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetSignalizedNodeRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.29 NetReadPhasingPlan

Signature: `int NetReadPhasingPlan(FILE * file, TNetPhasingPlanRecord * record)`

Description: Read a record from a phasing plan table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetPhasingPlanRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.30 NetWritePhasingPlan

Signature: `int NetWritePhasingPlan(FILE * file, const TNetPhasingPlanRecord * record)`

Description: Write a record to a phasing plan table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetPhasingPlanRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.31 NetReadTimingPlan

Signature: `int NetReadTimingPlan(FILE * file, TNetTimingPlanRecord * record)`

Description: Read a record from a timing plan table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetTimingPlanRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.32 NetWriteTimingPlan

Signature: `int NetWriteTimingPlan(FILE * file, const TNetTimingPlanRecord * record)`

Description: Write a record to a timing plan table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetTimingPlanRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.33 NetReadDetector

Signature: `int NetReadDetector(FILE * file, TNetDetectorRecord * record)`

Description: Read a record from a detector table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetDetectorRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.34 NetWriteDetector

Signature: `int NetWriteDetector(FILE * file, const TNetDetectorRecord * record)`

Description: Write a record to a detector table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetDetectorRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.35 NetReadSignalCoordinator

Signature: `int NetReadSignalCoordinator(FILE * file, TNetSignalCoordinatorRecord * record)`

Description: Read a record from a signal coordinator table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetSignalCoordinatorRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.36 NetWriteSignalCoordinator

Signature: `int NetWriteSignalCoordinator(FILE * file, const TNetSignalCoordinatorRecord * record)`

Description: Write a record to a signal coordinator table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetSignalCoordinatorRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.37 NetReadActivityLocation

Signature: `int NetReadActivityLocation(FILE * file, TNetActivityLocationRecord * record)`

Description: Read a record from an activity location table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetActivityLocationRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.38 NetWriteActivityLocation

Signature: `int NetWriteActivityLocation(FILE * file, const TNetActivityLocationRecord * record)`

Description: Write a record to a process link table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetActivityLocationRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.39 NetReadProcessLink

Signature: `int NetReadProcessLink(FILE * file, TNetProcessLinkRecord * record)`

Description: Read a record from a process link table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetProcessLinkRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.40 NetWriteProcessLink

Signature: `int NetWriteProcessLink(FILE * file, const TNetProcessLinkRecord * record)`

Description: Write a record to a process link table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetProcessLinkRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.3.41 NetReadStudyAreaLink

Signature: `int NetReadStudyAreaLink(FILE * file, TNetStudyAreaLinkRecord * record)`

Description: Read a record from a study area link table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetStudyAreaLinkRecord` structure into which the record is read.

Return Value: Return nonzero if the record was successfully read, or zero if not.

7.3.42 NetWriteStudyAreaLink

Signature: `int NetWriteStudyAreaLink(FILE * file, const TNetStudyAreaLinkRecord * record)`

Description: Write a record to a study area link table.

Argument: `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetStudyAreaLinkRecord` structure from which the record is written.

Return Value: Return nonzero if the record was successfully written, or zero if not.

7.4 Data Structures

7.4.1 TNetHeader

This structure is used for the network table header.

```
typedef struct
{
/** The field names. **/
INT8 fFields[512];

} TNetHeader;
```

7.4.2 TNetNodeRecord

This structure is used for network node table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The EASTING field. **/
REAL64 fEasting;

/** The NORTHING field.  **/
REAL64 fNorthing;

/** The ELEVATION field. **/
REAL64 fElevation;

/** The NOTES field. **/
INT8 fNotes[256];

} TNetNodeRecord
```

7.4.3 TNetLinkRecord

This structure isused for network link table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The NAME field. **/
INT8 fName[51];

/** The NODEA field. **/
INT32 fNodea;

/** The NODEB field. **/
INT32 fNodeb;

/** The PERMLANESA field. **/
```

```
INT32 fPermlanesa;

/** The PERMLANESB field. **/
INT32 fPermlanesb;

/** The LEFTPCKTSA field. **/
INT32 fLeftpcktsa;

/** The LEFTPCKTSB field.  **/
INT32 fLeftpcktsb;

/** The RIGHTPCKTSA field. **/
INT32 fRightpcktsa;

/** The RIGHTPCKTSB field. **/
INT32 fRightpcktsb;

/** The TWOWAYTURN field. **/
INT8 fTwowayturn[2];

/** The LENGTH field. **/
REAL64 fLength;

/** The GRADE field. **/
REAL64 fGrade;

/** The SETBACKA field. **/
REAL64 fSetbacka;

/** The SETBACKB field. **/
REAL64 fSetbackb;

/** The CAPACITYA field. **/
INT32 fCapacitya;

/** The CAPACITYB field. **/
INT32 fCapacityb;

/** The SPEEDLMTA field. **/
REAL64 fSpeedlmta;

/** The SPEEDLMTB field. **/
REAL64 fSpeedlmtb;

/** The FREESPDA field. **/
REAL64 fFreespda;

/**   The FREESPDB field. **/
REAL64 fFreespdb;

/** The FUNCTCLASS field. **/
INT8 fFunctclass[11];

/** The THRUA field. **/
INT32 fThrua;

/** The THRUB field. **/
INT32 fThrub;
```

```
/** The COLOR field. **/
INT32 fColor;

/** The VEHICLE field. **/
INT8 fVehicle[101];

/** the NOTES field. **/
INT8 fNotes[256];

} TNetNodeRecord;
```

7.4.4 TNetSpeedRecord

This structure is used for network speed table records.

```
typedef struct
{
/** The LINK field. **/
INT32 fLink;

/** The NODE field. **/
INT32 fNode;

/** The SPEEDLMT field. **/
REAL64 fSpeedlmt;

/**  The FREESPD field. **/
REAL64 fFreespd;

/** The VEHICLE field. **/
INT8 fVehicle[101];

/** The STARTTIME field. **/
INT8 fStarttime[9];

/** The ENDTIME field. **/
INT8 fEndtime[9];

/** The NOTES field. **/
INT8 fNotes[256]

} TNetSpeedRecord;
```

7.4.5 TNetPocketRecord

This structure is used for network pocket lane table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The NODE field. **/
INT32 fNode;

/** The LINK field. **/
INT32 fLink;
```

```
/** The OFFSET field. **/
REAL64 fOffset;

/** The LANE field. **/
INT32 fLane;

/** The STYLE field. **/
INT8 fStyle[2];

/** The LENGTH field. **/
REAL64 fLength;

/** The NOTES field. **/
INT8 fNotes[256];

} TNetPocketRecord;
```

7.4.6 TNetLaneUseRecord

This structure is used for network lane use table records.

```
typedef struct
{
/** The NODE field. **/
INT32 fNode;

/** The LINK field. **/
INT32 fLink;

/** The LANE field. **/
INT32 fLane;

/** The VEHICLE field. **/
INT8 fVehicle[101];

/** The RESTRICT field. **/
INT8 fRestrict[2];

/** The STARTTIME field. **/
INT8 fStarttime[9];

/** The ENDTIME field. **/
INT8 fEndtime[9];

/** The NOTES field. **/
INT8 fNotes[256];

} TNetLaneUseRecord
```

7.4.7 TNetParkingRecord

This structure is used for network parking table records.

```
typedef struct
{
/** The ID field. **/
```

```
INT32 fId;

/** The NODE field. **/
INT32 fNode;

/** The LINK field. **/
INT32 fLink;

/** The OFFSET field. **/
REAL64 fOffset;

/** The STYLE field. **/
INT8 fStyle[6];

/** The CAPACITY field. **/
INT32 fCapacity;

/** The GENERIC field. **/
INT8 fGeneric[2];

/** The VEHICLE field. **/
INT8 fVehicle[101];

/** The STARTTIME field. **/
INT8 fStarttime[9];

/** The ENDTIME field. **/
INT8 fEndtime[9];

/** The NOTES field. **/
INT8 fNotes[256];

} TNetParkingRecord;
```

7.4.8 TNetBarrierRecord

This structure is used for network barrier table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The NODE field. **/
INT32 fNode;

/** The LINK field. **/
INT32 fLink;

/** The OFFSET field. **/
REAL64 fOffset;

/** The LANE field. **/
INT32 fLane;

/** The STYLE field. **/
INT8 fStyle[11];
```

```
/** The LENGTH field. **/
REAL64 fLength;

/** The NOTES field. **/
INT8 fNotes[256];

} TNetBarrierRecord;
```

7.4.9 TNetTransitStopRecord

This structure is used for network transit stop table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The NAME field. **/
INT8 fName[51];

/** The NODE field. **/
INT32 fNode;

/** The LINK field. **/
INT32 fLink;

/** The OFFSET field. **/
REAL64 fOffset;

/** The VEHICLE field. **/
INT8 fVehicle[101];

/** The STYLE field. **/
INT8 fStyle[11];

/** The CAPACITY field. **/
INT32 fCapacity;

/** The NOTES field. **/
INT8 fNotes[256];

} TNetTransitStopRecord;
```

7.4.10 TNetLaneConnectivityRecord

This structure is used for network lane connectivity table records.

```
typedef struct
{
/** The NODE field. **/
INT32 fNode;

/** The INLINK field. **/
INT32 fInlink;

/** The INLANE field. **/
INT32 fInlane;
```

```
/** The OUTLINK field. **/
INT32 fOutlink;

/** The OUTLANE field. **/
INT32 fOutlane;

/** The NOTES field. **/
INT8 fNotes[256];

} TNetLaneConnectivityRecord;
```

7.4.11 TNetTurnProhibitionRecord

This structure is used for network turn prohibition table records.

```
typedef struct
{
/** The NODE field. **/
INT32 fNode;

/** The INLINK field. **/
INT32 fInlink;

/** The OUTLINK field. **/
INT32 fOutlink;

/** The STARTTIME field. **/
INT8 fStarttime[9];

/** The ENDTIME field. **/
INT8 fEndtime[9];

/** The NOTES field. **/
INT8 fNotes[256];

} TNetTurnProhibitionRecord;
```

7.4.12 TNetUnsignalizedNodeRecord

This structure is used for network unsignalized node table records.

```
typedef struct
{
/** The NODE field. **/
INT32 fNode;

/**  The INLINK field. **/
INT32 fInlink;

/** The SIGN field. **/
INT8 fSign[2];

/** The NOTES field. **/
INT8 fNotes;

} TNetUnsignalizedNodeRecord;
```

7.4.13 TNetSignalizedNodeRecord

This structure is used for network signalized node table records.

```
typedef struct
{
/** The NODE field. **/
INT32 fNode;

/** The TYPE field. **/
INT8 fType[2];

/** The PLAN field. **/
INT32 fPlan;

/** The OFFSET field. **/
REAL64 fOffset;

/** The STARTTIME field. **/
INT8 fStarttime[9];

/** The COORDINATR field. **/
INT32 fCoordinatr;

/** The RING field. **/
INT8  fRing[2];

/** The ENTRY field. **/
INT8 fEntry[2];

/** The NOTES field. **/
INT8 fNotes[256];

} TNetSignalizedNodeRecord;
```

7.4.14 TNetPhasingPlanRecord

This structure is used for network phasing plan table records.

```
typedef struct
{
/** The NODE field. **/
INT32 fNode;

/** The PLAN field. **/
INT32 fPlan;

/** The PHASE field. **/
INT32 fPhase;

/** The INLINK field. **/
INT32 fInlink;

/** The OUTLINK field. **/
INT32 fOutlink;

/** The PROTECTION field. **/
INT8 fProtection[2];
```

```
/** The NOTES field. **/
INT8 fNotes[256];

} TNetPhasingPlanRecord;
```

7.4.15 TNetTimingPlanRecord

This structure is used for network timing plan table records.

```
typedef struct
{
/** The PLAN field. **/
INT32 fPlan;

/** The PHASE field. **/
INT32 fPhase;

/** The NEXTPHASES field. **/
INT8 fNextphases[21];

/** The GREENMIN field. **/
REAL64 fGreenmin;

/** The GREENMAX field. **/
REAL64 fGreenmax;

/** The GREENEXT field. **/
REAL64 fGreenext;

/** The YELLOW field. **/
REAL64 fYellow;

/** The REDCLEAR field. **/
REAL64 fRedclear;

/** The GROUPFIRST field. **/
INT32 fGroupfirst;

/** The NOTES field. **/
INT8 fNotes[256];

} TNetTimingPlanRecord;
```

7.4.16 TNetDetectorRecord

This structure is used for network detector table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The NODE field. **/
INT32 fNode;

/** The LINK field. **/
```

```
INT32 fLink;

/** The OFFSET field. **/
REAL64 fOffset;

/** The LANEBEGIN field. **/
INT32 fLanebegin;

/** The LANEEND field. **/
INT32 fLaneend;

/** The LENGTH field. **/
REAL64 fLength;

/** The STYLE field. **/
INT8 fStyle[11];

/** The COORDINATR field. **/
INT8 fCoordinatr[51];

/** The NOTES field. **/
INT8 fNotes[256];

} TNetDetectorRecord;
```

7.4.17 TNetSignalCoordinatorRecord

This structure is used for network signal coordinator table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The TYPE field. **/
INT8 fType[11];

/** The ALGORITHM field. **/
INT8 fAlgorithm[11];

/** The NOTES field. **/
INT8 fNotes;

} TNetSignalCoordinatorRecord;
```

7.4.18 TNetActivityLocationRecord

This structure is used for activity location table records.

```
/** Maximum allowed optional user-defined fields in activity location data.
**/
#define ACTIVITY_MAX_USER 20

typedef struct
{
/** The ID field. **/
INT32 fId;
```

```
/** The NODE field. **/
INT32 fNode;

/** The LINK field. **/
INT32 fLink;

/** The OFFSET field. **/
REAL64 fOffset;

/** The LAYER field. **/
INT8 fLayer[11];

/** The EASTING field. **/
REAL64 fEasting;

/** The NORTHING field. **/
REAL64 fNorthing;

/** The ELEVATION field. **/
REAL64  fElevation;

/** The number of values in the fUserName and fUser Data arrays.
INT32 fNumberUser;

/** Optional array of user-defined real values.  The number of values
/*  in the array is variable, but must be the same in each record.
/*  The data will typically be related to land use.
/*  The optional fields immediately precede the NOTES field. **/
REAL64 fUserData[ACTIVITY_MAX_USER];

/** The names of the fields in fUser Data. **/
INT8 fUserNames[ACTIVITY_MAX_USER] [32];

/** The NOTES field. **/
INT8 fNotes[256];

} TNetActivityLocationRecord;
```

7.4.19 TNetProcessLinkRecord

This structure is used for process link table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The FROMID field. **/
INT32 fFromid;

/** The FROMTYPE field. **/
INT8 fFromtype[11];

/** The TOID field. **/
INT32 fToid;

/** The TOTYPE field. **/
```

```
INT8  fTotype[11];

/** The DELAY field. **/
REAL64 fDelay;

/** The COST field. **/
REAL64 fCost;

/** The NOTES field. **/
INT8 fNotes[256];

} TNetProcessLinkRecord;
```

7.4.20 TNetStudyAreaLinkRecord

This structure is used for network study area link table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The BUFFER field. **/
INT8  fBuffer[2];

/** The NOTES field. **/
INT8 fNotes;

} TNetStudyAreaLinkRecord;
```

7.5 Utility Programs

Several utility programs related to network data files are available.

7.5.1 ReadNetwork

The *ReadNetwork* application reads a specified set of network tables into memory and constructs C++ network objects out of it. It is useful for verifying that a network can be read by the route planner and microsimulation without actually running those programs. It takes a configuration file as its only argument.

7.5.2 ValidateNetwork

The *ValidateNetwork* application reads a specified set of network tables into memory and looks for errors, inconsistencies, and suspicious data in them. It is useful for checking the validity of network data files before using them in a simulation. It takes a configuration file and a log (output) file as its two arguments. The configuration file key NET_VALIDATE_WARNINGS should be set to zero or one, depending upon whether the tool should list warnings in addition to errors.

7.5.3 SetupNetwork

The *SetupNetwork* script copies a set of empty and test network tables into a specified directory. It is useful for building a new network database directory. It takes the name of the directory as its only argument.

7.5.4 CleanupNetwork

The *CleanupNetwork* script removes a set of tables created by *SetupNetwork*. It takes the name of the directory as its argument.

7.6 Files

Table 43: Network library files.

Type	File Name	Description
Binary Files	libTIO.a	TRANSIMS Interfaces library
Utilities	ReadNetwork	Network data file reader
	ValidateNetwork	Network data file validator
	SetupNetwork	Tool for creating empty and test network data files
	CleanupNetwork	Tool for removing empty and test network data files
Source Files	netio.c	Defines network data structures and interface functions
	netio.h	Network interface functions source file
Example Files	Test*.tbl	Test network tables
	Test.config	Configuration file for test network

7.7 Configuration Keys

Table 44 below lists the TRANSIMS configuration file keys that specify the location of network data files.

Table 44: Network file configuration keys.

Configuration Key	Description
NET_DIRECTORY	Directory where the network files reside.
NET_NODE_TABLE	Node table name.
NET_LINK_TABLE	Link table name.
NET_POCKET_LANE_TABLE	Pocket lane table name.
NET_PARKING_TABLE	Parking table name.
NET_LANE_CONNECTIVITY_TABLE	Lane connectivity table name.
NET_UNSIGNALIZED_NODE_TABLE	Unsignalized node table name.
NET_SIGNALIZED_NODE_TABLE	Signalized node table name.
NET_PHASING_PLAN_TABLE	Phasing plan table name.
NET_TIMING_PLAN_TABLE	Timing plan table name.
NET_SPEED_TABLE	Speed table name.
NET_LANE_USE_TABLE	Lane use table name.
NET_TRANSIT_STOP_TABLE	Transit stop table name.
NET_SIGNAL_COORDINATOR_TABLE	Signal coordinator table name.
NET_DETECTOR_TABLE	Detector table name.

Configuration Key	Description
NET_TURN_PROHIBITION_TABLE	Turn prohibition table name.
NET_BARRIER_TABLE	Barrier table name.
NET_ACTIVITY_LOCATION_TABLE	Activity location table name.
NET_PROCESS_LINK_TABLE	Process link table name.
NET_STUDY_AREA_LINKS_TABLE	Study area links table name.
NET_LINK_MEDIAN_HALFWIDTH	Default half-width (meters) of the median between lanes on a link. ☛ To correspond with the IOC-2 TRANSIMS Visualization tool, this parameter must be assigned a value of 0.5 * GBL_LANE_WIDTH.

7.8 Examples

Figure 2 shows the layout of the network used for testing various TRANSIMS modules and Figure 3 gives the configuration file for the network. This network contains most of the network objects available in TRANSIMS; it can be used for testing code or in simulations of traffic. Table 44 lists the configuration keys for this network and Table 45 through Table 62 list the contents of the tables.

Figure 2: Layout of test network.

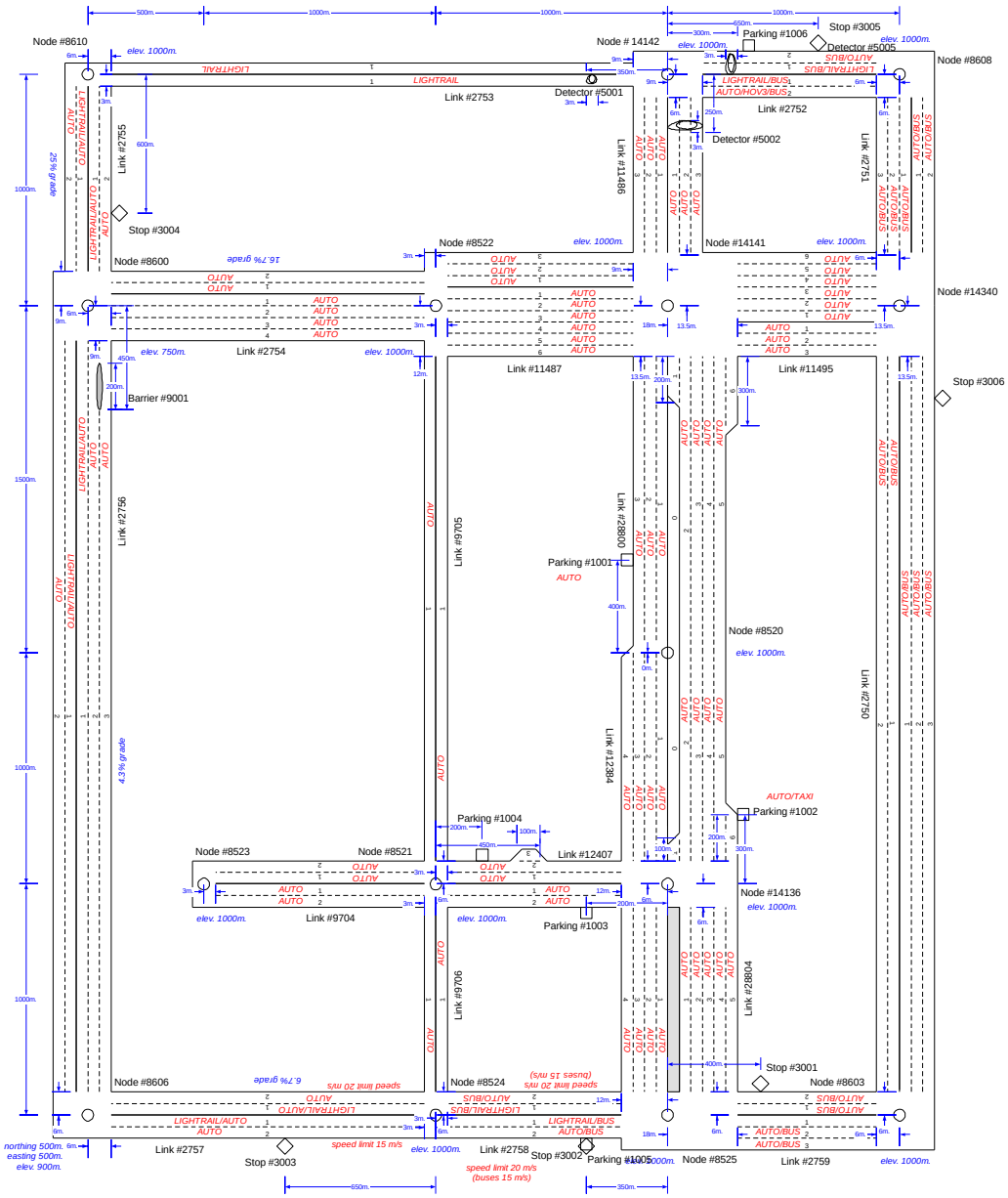
Node #8610
Node # 14142
Parking #1006
Stop #3005
Detector #5005
Node #8608
Link #2753
Detector #5001
Link #2752
Detector #5002
Link #2755
Link #11486
Link #2751
Stop #3004
Node #8522
Node #14141
Node #8600
Node #14340
Link #2754
Link #11487
Link #11495
Stop #3006
Barrier #9001
Link #2756
Link #9705
Link #28800
Parking #1001
Node #8520
Link #2750
Link #12384
Parking #1002
Parking #1004
Link #12407
Node #8523
Node #8521
Node #14136
Link #9704
Parking #1003
Link #9706
Link #28804
Stop #3001
Node #8606
Node #8524
Node #8603
Link #2757
Stop #3003
Link #2758
Stop #3002
Parking #1005
Node #8525
Link #2759
northing 500m.
easting 500m.
elev. 900m.

Figure 3: Test network configuration file.

```
# Network subsystem configuration keys for the test network.

# The directory where the network files reside.
NET_DIRECTORY /home/projects/transims/networks/test

# The node table name.
NET_NODE_TABLE Test_Node_Table

# The link table name.
NET_LINK_TABLE Test_Link_Table

# The pocket lane table name.
NET_POCKET_LANE_TABLE Test_Pocket_Lane_Table

# The parking table name
NET_PARKING_TABLE Test_Parking_Table

# The lane connectivity table name.
NET_LANE_CONNECTIVITY_TABLE Test_Lane_Connectivity_Table

# The unsignalized node table name.
NET_UNSIGNALIZED_NODE_TABLE Test_Unsignalized_Node_Table

# The signalized node table name.
NET_SIGNALIZED_NODE_TABLE Test_Signalized_Node_Table

# The phasing plan table name.
NET_PHASING_PLAN_TABLE Test_Phasing_Plan_Table

# The timing plan table name.
NET_TIMING_PLAN_TABLE Test_Timing_Plan_Table

# The speed table name.
NET_SPEED_TABLE Test_Speed_Table

# The lane use table name.
NET_LANE_USE_TABLE Test_Lane_Use_Table

# The transit stop table name.
NET_TRANSIT_STOP_TABLE Test_Transit_Stop_Table

# The signal coordinator table name.
NET_SIGNAL_COORDINATOR_TABLE Test_Signal_Coordinator_Table

# The detector table name.
NET_DETECTOR_TABLE Test_Detector_Table

# The turn prohibition table name.
NET_TURN_PROHIBITION_TABLE Test_Turn_Prohibition_Table

# The barrier table name.
NET_BARRIER_TABLE Test_Barrier_Table

# The activity location table name.
NET_ACTIVITY_LOCATION_TABLE Test_Activity_Location_Table

# The process link table name.
NET_PROCESS_LINK_TABLE Test_Process_Link_Table

# The study area links table name.
NET_STUDY_AREA_LINKS_TABLE Test_Study_Area_Link_Table

# The half-width (meters) of the median on a link.
# Default value if this keyword is omitted is 0.5 * GBL_LANE_WIDTH NET_LINK_MEDIAN_HALFWIDTH     1.75
```

Table 45: Test node table.

ID	EASTING	NORTHING	ELEVATION	NOTES
8520	3000	2500	1000	
8521	2000	1500	1000	
14136	3000	1500	1000	
14141	3000	4000	1000	
14142	3000	5000	1000	
14340	4000	4000	1000	
8525	3000	500	1000	
8522	2000	4000	1000	
8523	1000	1500	1000	
8524	2000	500	1000	
8606	500	500	900	
8603	4000	500	1000	
8608	4000	5000	1000	
8600	500	4000	750	
8610	500	5000	1000	

Table 46: Test link table.

ID	NAME	NODEA	NODEB	PERMLANESA	PERMLANESB	LEFTPCKTSA	LEFTPCKTSB	RGHTPCKTSA	RGHTPCKTSB	TWOWAYTURN	LENGTH	GRADE	SETBACKA	SETBACKB	CAPACITYA	CAPACITYB	SPEEDLMTA	SPEEDLMTB	FREESPDA	FREESPDB	FUNC
9704	2nd Street	8521	8523	2	2	0	0	0	0	F	1000	0	3	3	800	1000	20	20	25	25	OTHE
9705	Avenue B	8521	8522	1	1	0	0	0	0	F	2500	0	6	12	800	1000	20	20	25	25	ZONE
9706	Avenue B	8521	8524	1	1	0	0	0	0	F	1000	0	6	6	800	1000	20	20	25	25	RAMI
11486	Avenue C	14141	14142	3	3	0	0	0	0	F	1000	0	13.5	6	800	1000	20	20	25	25	FRON
11487	3rd Street	8522	14141	3	6	0	0	0	0	F	1000	0	3	9	800	1000	20	20	25	25	SECA
11495	3rd Street	14141	14340	6	3	0	0	0	0	F	1000	0	18	6	800	1000	20	20	25	25	COLL
12384	Avenue C	14136	8520	4	4	0	1	0	1	T	1000	0	6	0	800	1000	20	20	25	25	FREE
12407	2nd Street	8521	14136	2	2	0	0	1	0	F	1000	0	3	12	500	500	20	20	25	25	XPRE
28800	Avenue C	8520	14141	3	4	0	1	0	1	T	1500	0	0	13.5	800	1000	20	20	25	25	PRIAR
28804	Avenue C	14136	8525	5	4	0	0	0	0	F	1000	0	6	6	800	1000	20	20	25	25	LOCA
2759	1st Street	8525	8603	2	3	0	0	0	0	F	1000	0	18	6	800	1000	20	20	25	25	LOCA
2750	Avenue D	8603	14340	2	3	0	0	0	0	F	3500	0	6	13.5	800	1000	20	20	25	25	LOCA
2751	Avenue D	14340	8608	3	2	0	0	0	0	F	1000	0	13.5	6	800	1000	20	20	25	25	LOCA
2752	4th Street	8608	14142	2	2	0	0	0	0	F	1000	0	6	9	800	1000	20	20	25	25	LOCA
2753	4th Street	14142	8610	1	1	0	0	0	0	F	2500	0	9	6	800	1000	20	20	25	25	LIGH
2755	Avenue A	8610	8600	2	2	0	0	0	0	F	1000	-25	3	9	800	1000	20	20	25	25	LOCA
2754	3rd Street	8600	8522	2	4	0	0	0	0	F	1500	16.7	6	3	800	1000	20	20	25	25	LOCA
2756	Avenue A	8600	8606	3	2	0	0	0	0	F	3500	4.3	9	6	800	1000	20	20	25	25	LOCA
2757	1st Street	8606	8524	2	2	0	0	0	0	F	1500	6.7	6	3	800	1000	20	20	25	25	LOCA
2758	1st Street	8524	8525	2	2	0	0	0	0	F	1000	0	3	12	800	1000	20	20	25	25	LOCA

Table 47: Test speed table.

LINK	NODE	SPEEDLMT	FREESPD	VEHICLE	STARTTIME	ENDTIME	NOTES
2758	8524	15	20	BUS	ALL00:00	ALL24:00	
2758	8525	15	18	BUS	ALL00:00	ALL24:00	

Table 48: Test pocket lane table.

ID	NODE	LINK	OFFSET	LANE	STYLE	LENGTH	NOTES
85201	8520	12384	0	1	M	100	
85206	8520	12384	0	6	M	200	
85213	8521	12407	450	3	P	100	
141411	14141	28800	0	1	T	200	
141416	14141	28800	0	6	T	300	

Table 49: Test lane use table.

NODE	LINK	LANE	VEHICLE	RESTRICT	STARTTIME	ENDTIME	NOTES
8606	2757	2	AUTO/HOV3	O	ALL00:00	ALL24:00	
8524	2757	1	LIGHTRAIL	R	ALL00:00	ALL24:00	
8524	2758	1	LIGHTRAIL	R	ALL00:00	ALL24:00	
8524	2758	2	AUTO	R	ALL00:00	ALL24:00	
8525	2758	1	AUTO	N	ALL00:00	ALL24:00	
8525	2758	2	LIGHTRAIL	N	ALL00:00	ALL24:00	
8606	2756	1	LIGHTRAIL	R	ALL00:00	ALL24:00	
8600	2756	1	LIGHTRAIL	R	ALL00:00	ALL24:00	
8600	2755	2	LIGHTRAIL	N	ALL00:00	ALL24:00	
8610	2755	1	LIGHTRAIL	R	ALL00:00	ALL24:00	
14142	2752	1	LIGHTRAIL	R	ALL00:00	ALL24:00	
14142	2752	2	AUTO	R	ALL00:00	ALL24:00	
8608	2752	1	AUTO	N	ALL00:00	ALL24:00	
8608	2752	1	LIGHTRAIL	R	ALL00:00	ALL24:00	

Table 50: Test parking table.

ID	NODE	LINK	OFFSET	STYLE	CAPACITY	GENERIC	VEHICLE	STARTTIME	ENDTIME	NOTES
1001	8520	28800	400	LOT	50	T	AUTO	ALL00:00	ALL24:00	
1002	14136	12384	300	PRSTR	10	T	AUTO/TAXI	ALL00:00	ALL24:00	
1003	14136	12407	200	HISTR	10	T	ANY	ALL00:00	ALL24:00	
1004	8521	12407	200	DRVWY	1	F	ANY	ALL00:00	ALL24:00	
1005	8525	2758	370	LOT	1	F	BUS	ALL00:00	ALL24:00	
1006	14142	2752	650	LOT	0	F	ANY	ALL00:00	ALL24:00	

Table 51: Test barrier table.

ID	NODE	LINK	OFFSET	LANE	STYLE	LENGTH	NOTES
9001	8600	2756	450	1	BARRIER	200	

Table 52: Test transit stop table.

ID	NAME	NODE	LINK	OFFSET	VEHICLE	STYLE	CAPACITY	NOTES
3001	1st & C NE	8525	2759	400	BUS	STOP	25	
3002	1st & C SW	8525	2758	350	BUS/LIGHTRAIL	STATION	0	
3003	1st & B	8524	2757	650	LIGHTRAIL	YARD	0	
3004	4th & A	8610	2755	600	LIGHTRAIL	STOP	200	
3005	4th & C	14142	2752	650	BUS/LIGHTRAIL	STATION	0	
3006	3rd & D	14340	2750	400	BUS	STOP	1	

Table 53: Test lane connectivity table.

NODE	INLINK	INLANE	OUTLINK	OUTLANE	NOTES
14141	11487	1	11486	1	
14141	11487	2	11486	2	
14141	11487	3	11495	1	
14141	11487	4	11495	2	
14141	11487	5	11495	3	
14141	11487	6	28800	3	
14141	11486	1	11495	1	
14141	11486	2	28800	1	
14141	11486	3	28800	2	
14141	11486	3	11487	3	
14141	11495	1	28800	1	
14141	11495	2	28800	2	
14141	11495	3	11487	1	
14141	11495	4	11487	2	
14141	11495	5	11487	3	
14141	11495	6	11486	3	
14141	28800	1	11487	1	
14141	28800	2	11487	2	
14141	28800	3	11486	1	
14141	28800	4	11486	2	
14141	28800	5	11486	3	
14141	28800	6	11495	3	
8520	12384	2	28800	2	
8520	12384	3	28800	3	
8520	12384	4	28800	4	
8520	12384	5	28800	5	
8520	28800	1	12384	1	
8520	28800	2	12384	2	
8520	28800	3	12384	3	
8520	28800	3	12384	4	
14136	12407	1	12384	1	
14136	12407	2	28804	4	
14136	12384	1	28804	1	
14136	12384	2	28804	2	
14136	12384	3	28804	3	
14136	12384	4	28804	4	
14136	12384	4	12407	2	
14136	28804	1	12407	1	
14136	28804	1	12384	2	
14136	28804	2	12384	3	
14136	28804	3	12384	4	
14136	28804	4	12384	5	

NODE	INLINK	INLANE	OUTLINK	OUTLANE	NOTES
14136	28804	5	12384	6	
8521	12407	1	9704	1	
8521	12407	1	9706	1	
8521	12407	2	9704	2	
8521	12407	2	9705	1	
8521	9704	1	12407	1	
8521	9704	1	9705	1	
8521	9704	2	12407	2	
8521	9704	2	9706	1	
8521	9705	1	9706	1	
8521	9705	1	9704	2	
8521	9705	1	12407	1	
8521	9706	1	9705	1	
8521	9706	1	12407	2	
8521	9706	1	9704	1	
14340	2750	1	11495	1	
14340	2750	2	11495	2	
14340	2750	2	11495	3	
14340	2750	3	2751	1	
14340	2750	3	2751	2	
14340	11495	1	2751	1	
14340	11495	2	2751	2	
14340	11495	2	2750	1	
14340	11495	3	2750	2	
14340	2751	1	2750	1	
14340	2751	1	11495	4	
14340	2751	2	2750	2	
14340	2751	2	11495	5	
14340	2751	3	11495	6	
8608	2751	1	2752	1	
8608	2751	2	2752	2	
8608	2752	1	2751	1	
8608	2752	1	2751	2	
8608	2752	2	2751	3	
8603	2759	1	2750	1	
8603	2759	2	2750	2	
8603	2759	3	2750	3	
8603	2750	1	2759	1	
8603	2750	2	2759	2	
8606	2757	1	2756	1	
8606	2757	1	2756	2	
8606	2757	2	2756	3	
8606	2756	1	2757	1	
8606	2756	2	2757	2	
8610	2753	1	2755	1	
8610	2755	1	2753	1	
8600	2756	1	2755	1	
8600	2756	2	2755	2	
8600	2756	2	2754	3	
8600	2756	3	2754	4	
8600	2754	1	2756	1	
8600	2754	2	2755	2	
8600	2755	1	2756	1	
8600	2755	2	2756	2	
8600	2755	1	2754	1	

NODE	INLINK	INLANE	OUTLINK	OUTLANE	NOTES
14142	2753	1	2752	1	
14142	11486	3	2752	2	
14142	2752	1	2753	1	
14142	2752	2	11486	1	
8522	2754	1	11487	2	
8522	2754	2	11487	3	
8522	2754	3	11487	4	
8522	2754	4	11487	5	
8522	2754	4	9705	1	
8522	9705	1	11487	6	
8522	11487	1	9705	1	
8522	11487	2	2754	1	
8522	11487	3	2754	2	
8524	2758	1	2757	1	
8524	2758	2	2757	2	
8524	2758	2	9706	1	
8524	9706	1	2757	2	
8524	9706	1	2758	2	
8524	2757	1	2758	1	
8524	2757	2	2758	2	
8524	2757	1	9706	1	
8525	2758	1	2759	2	
8525	2758	2	2759	3	
8525	2759	1	2758	1	
8525	2759	2	2758	2	
8525	2759	2	28804	5	
8525	2758	2	28804	1	
8525	28804	1	2759	1	
8525	28804	2	2759	2	
8525	28804	3	2759	3	
8525	28804	4	2758	2	

Table 54: Test unsignalized node table.

NODE	INLINK	SIGN	NOTES
8520	12384	Y	
8520	28800	N	
14136	12407	S	
14136	12384	N	
14136	28804	N	
8610	2753	N	
8610	2755	N	
14142	2753	N	
14142	11486	S	
14142	2752	N	
8608	2751	N	
8608	2752	N	
8600	2756	N	
8600	2754	S	
8600	2755	N	
8522	2754	N	
8522	9705	S	
8522	11487	N	
14340	2751	S	
14340	11495	S	
14340	2750	S	
8606	2756	N	
8606	2757	N	
8524	2757	N	
8524	2758	N	
8524	9706	Y	
8525	2758	N	
8525	2759	N	
8525	28804	S	
8603	2759	N	
8603	2750	N	

Table 55: Test signalized node table.

NODE	TYPE	PLAN	OFFSET	STARTTIME	COORDINATR	RING	ENTRY	NOTES
14141	T	1	19	ALL00:00	0	S	S	
8521	A	2	0	ALL18:00	0	S	S	
8521	A	3	0	WKD07:00	0	S	S	

Table 56: Test phasing plan table.

NODE	PLAN	PHASE	INLINK	OUTLINK	PROTECTION	NOTES
14141	1	1	11487	11495	U	
14141	1	1	11487	28800	P	
14141	1	1	11495	11487	U	
14141	1	1	11495	11486	P	
14141	1	1	11486	11487	S	
14141	1	1	28800	11495	S	
14141	1	2	11487	28800	P	
14141	1	2	11495	11486	P	
14141	1	2	11486	11495	P	
14141	1	2	28800	11487	P	
14141	1	2	28800	11495	S	
14141	1	2	11486	11487	S	
14141	1	3	11487	28800	P	
14141	1	3	28800	11487	P	
14141	1	3	28800	11486	U	
14141	1	3	28800	11495	P	
14141	1	3	11495	11486	S	
14141	1	3	11486	11487	S	
14141	1	4	11487	28800	P	
14141	1	4	11486	11495	U	
14141	1	4	11486	28800	U	
14141	1	4	11486	11487	P	
14141	1	4	28800	11486	U	
14141	1	4	28800	11495	P	
14141	1	4	11495	11486	S	
14141	1	5	11487	11486	P	
14141	1	5	11487	28800	P	
14141	1	5	11495	28800	P	
14141	1	5	11495	11486	S	
14141	1	5	11486	11487	P	
14141	1	5	28800	11495	P	
14141	1	6	11487	28800	P	
14141	1	6	11495	28800	P	
14141	1	6	11495	11487	U	
14141	1	6	11495	11486	P	
14141	1	6	11486	11487	S	
14141	1	6	28800	11495	P	
8521	2	1	9705	9704	U	
8521	2	1	9705	9706	U	
8521	2	1	9705	12407	U	
8521	2	1	9706	9705	U	
8521	2	1	9706	12407	U	
8521	2	1	9706	9704	U	
8521	2	2	12407	9704	U	
8521	2	2	12407	9705	U	
8521	2	2	12407	9706	U	
8521	2	2	9704	12407	U	
8521	2	2	9704	9705	U	
8521	2	2	9704	9706	U	
8521	3	1	9705	9704	U	
8521	3	1	9705	9706	U	
8521	3	1	9705	12407	U	

NODE	PLAN	PHASE	INLINK	OUTLINK	PROTECTION	NOTES
8521	3	1	9706	9705	U	
8521	3	1	9706	12407	U	
8521	3	1	9706	9704	U	
8521	3	2	12407	9706	P	
8521	3	2	9704	9705	P	
8521	3	3	12407	9704	U	
8521	3	3	12407	9705	U	
8521	3	3	12407	9706	U	
8521	3	3	9704	12407	U	
8521	3	3	9704	9705	U	
8521	3	3	9704	9706	U	

Table 57: Test timing plan table.

PLAN	PHASE	NEXTPHASES	GREENMIN	GREENMAX	GREENEXT	YELLOW	REDCLEAR	GROUPFIRST	NOTES
1	1	2	35	0	0	4	0	1	
1	2	3	5	0	0	3	0	0	
1	3	4	8	0	0	3	0	0	
1	4	5	32	0	0	4	0	0	
1	5	6	9	0	0	3	0	0	
1	6	1	1	0	0	3	0	0	
2	1	2	12	30	4	3	0	1	
2	2	1	10	40	4	3	0	0	
3	1	2	12	30	4	3	1	1	
3	2	3	4	8	2	3	0	0	
3	3	1	10	20	4	3	1	0	

Table 58: Test detector table.

ID	NODE	LINK	OFFSET	LANEBEGIN	LANEEND	LENGTH	STYLE	COORDINATR	NOTES
5001	14142	2753	350	1	1	3	PASSAGE	1000	
5002	14142	11486	250	1	3	3	PRESENCE	1000	
5005	14142	2752	300	1	2	3	PASSAGE	1000	

Table 59: Test signal coordinator table.

ID	TYPE	ALGORITHM	NOTES
1000			

Table 60: Test activity location table.

ID	NODE	LINK	OFFSET	LAYER	EASTING	NORTHING	ELEVATION	ACCESS	HOME	WORK	NOTES
23	8524	9706	200	AUTO	2000	700	1000	0.00	1.0	0.0	
24	8521	12407	300	BUS	2300	1500	1000	375.	0.0	1.0	

Table 61: Test process link table.

ID	FROMID	FROMTYPE	TOID	TOTYPE	DELAY	COST	NOTES
123	3003	TRANSIT	23	ACTIVITY	10	20	
124	24	ACTIVITY	1003	PARKING	30	40	

Table 62: Test study area link table.

ID	BUFFER	NOTES
9704	N	
9705	N	
9706	N	
11486	N	
11487	N	
11495	N	
12384	N	
12407	N	
28800	N	
28804	N	
2759	Y	
2750	Y	
2751	Y	
2752	Y	
2753	Y	
2755	Y	
2754	Y	
2756	Y	
2757	Y	
2758	Y	

8. SIMULATION OUTPUT

This TRANSIMS Simulation Output subsystem collects data from a running microsimulation and stores it for subsequent examination by the analyst or use by other TRANSIMS software components. It provides a software layer that insulates applications from the details of the file structure and provides great flexibility in the specification of the data to be collected.

Two very different modes of data collection are supported. The original mode, used in IOC-1, collects data in binary format on the local file system of each CPN used by the simulation. This data is postprocessed to merge it into a single file for analysis. The newer mode developed for IOC-2 uses a parallel communication library to collect data in ASCII format into a single file written by the master simulation process. No postprocessing is required with this mechanism.

8.1 Terms

Event Data Event data reports when an interesting event occurs for a traveler. Events are recorded as they occur, at irregular time intervals.

Evolution Data Evolution data, also known as snapshot data, provides detailed information about how the state of the simulation evolves in time. Evolution data may be recorded on every timestep or less frequently, as desired.

Summary Data Summary data reports aggregate data about the simulation. Summary data is sampled, accumulated, and reported periodically throughout the simulation.

8.2 File Format

This section describes the file formats of each of the eight types of simulation outputs currently implemented. All fields are described, but the filtering capability described in Section 8.3 allows suppression of any output field for which the analyst has no interest, thus resulting in smaller output files. Applications that read the output produced by the simulation should always use the functions for reading that are described in Section 8.3. The functions provided by the output representation automatically handle records with suppressed fields and only attempt to read the fields that were actually written. This enables the implementation of general postprocessing applications that need not be cognizant of the number and order of the fields written by the simulation.

8.2.1 Traveler Event

Traveler event records are output by the microsimulation each time an event that is of interest to the analyst occurs for a traveler. The simulation time interval during which to record events is defined in the input configuration file. Filtering capabilities are provided so that the analyst may choose which of the many potentially interesting events should be recorded. The events that may be of interest are specified in the STATUS and ANOMALY output fields in Table 63. The other fields describe the traveler's state at the time the event occurred.

Table 63: Traveler event record fields.

Field	Description
TIME	Current time (seconds from midnight).
TRAVELER	Traveler ID.
TRIP	Traveler's trip ID.
LEG	Traveler's plan leg ID.
VEHICLE	Vehicle ID; value = 0 if not in a vehicle.
VEHTYPE	Vehicle type: 0 = walk 1 = auto 2 = truck 3 = bicycle 4 = taxi 5 = bus 6 = trolley 7 = streetcar 8 = light rail 9 = rapid rail 10 = regional rail
VSUBTYPE	Vehicle subtype may be unused; value = 0 if not applicable.
ROUTE	Transit route ID; value = -1 if not in a transit vehicle.
STOPS	Count of number of stop signs encountered on current plan leg.
YIELDS	Count of number of yield signs encountered on current plan leg.
SIGNALS	Number of traffic signals encountered on current plan leg.
TURN	Type of last turn made: 0 = straight direction (no turn) 1 = right turn -1 = left turn 2 = hard right turn -2 = hard left turn values 3 to 6 represent increasingly more extreme right turns values –3 to –6 represent increasingly more extreme left turns -7 = reverse direction (U-turn)
STOPPED	Time (seconds) spent stopped on current plan leg.
ACCELS	Time (seconds) spent accelerating from 0 on current plan leg.
TIMESUM	Total time (seconds) spent on current plan leg.
DISTANCESUM	Total distance (meters) traveled on current plan leg (see accompanying text for more information).
USER	Analyst-defined field: any integer value is acceptable, and definition may vary with each case study.
ANOMALY	Type of anomaly: 0 = no anomaly occurred 1 = traveler is off plan 2 = traveler cannot find next link in plan 3 = traveler cannot find next parking place in plan 4 = traveler cannot find next vehicle in plan 5 = traveler cannot find next transit stop in plan 6 = traveler cannot board full transit vehicle 7 = driver of transit vehicle skipped stop that had passengers waiting to board 8 = driver of vehicle cannot change lanes because of congestion

Field	Description
STATUS	Traveler's current status bits: (see accompanying text for a detailed explanation of status bit interpretation). 0x1 = traveler is on a link (persistent) 0x2 = change in traveler's on-link status 0x4 = traveler is on a leg (persistent) 0x8 = change in traveler's on-leg status 0x10 = change in traveler's on-trip status 0x20 = traveler is non-motorized, i.e., walking, bicycling (persistent) 0x40 = traveler is not in the study area (persistent) 0x80 = change in traveler's in-study area status 0x100 = traveler is in a vehicle (persistent) 0x200 = change in traveler's vehicle occupancy status 0x400 = traveler is the driver (persistent) 0x800 = change in traveler's driver status 0x1000 = traveler is waiting at some location (persistent) 0x2000 = change in traveler's waiting status 0x4000 = location is a parking place (persistent) 0x8000 = location is a transit stop (persistent) 0x10000 = driver of transit vehicle is at a transit stop (persistent) 0x20000 = change in driver's transit vehicle at stop status 0x40000 = driver of transit vehicle is on a layover (persistent) 0x80000 = change in driver's transit vehicle on layover status 0x100000 = driver's transit vehicle is full (persistent) 0x200000 = change in driver's transit vehicle full status 0x400000 = traveler is off plan (persistent) 0x800000 = change in traveler's off-plan status 0x1000000 = beginning of simulation 0x2000000 = end of simulation 0x4000000 = location is an activity location (persistent) 0x8000000 = undefined 0x10000000 = undefined 0x20000000 = undefined 0x40000000 = undefined 0x80000000 = undefined

Field	Description
LOCATION	Where traveler is located: link ID, parking place ID, transit stop ID, or activity location ID, depending on the event as defined here EVENT — LOCATION value Enter/Exit/On link — link ID Begin/End plan leg — parking place ID or transit stop ID Begin/End trip — parking place ID or transit stop ID Enter/Exit study area — link ID Enter/Exit vehicle — parking place ID or transit stop ID Begin/End driving — parking place ID or transit stop ID Waiting for transit — transit stop ID Waiting at parking — parking place ID Begin/End activity — activity location ID Transit vehicle at stop — transit stop ID Transit vehicle on layover — transit stop ID Transit vehicle full — transit stop ID Off plan — link ID Begin/End Simulation — link ID Can't find link — link ID Can't find parking — parking place ID Can't find vehicle — parking place ID Can't find transit stop — transit stop ID Can't board transit — transit stop ID Skipped transit stop — transit stop ID Can't change lanes — link ID

The STATUS field is bit-oriented. Each bit represents a characteristic about the traveler that is true whenever the bit is set. Multiple bits set means that multiple characteristics are true at this time. Interpretation of the STATUS field involves determining which combination of characteristics is currently true according to the table that describes the individual bits. It is convenient to view the STATUS field in hexadecimal notation as this more clearly illuminates the patterns in the field.

Status values are generally represented in bit pairs. The lower bit of a pair is termed the *persistent* bit, and the upper bit is termed the *change bit*. The persistent bit is set during the entire time that the condition is true. The change bit is set only for the timestep when a change in the persistent bit occurs. This scheme allows the analyst to identify the beginning and end of a persistent condition without comparing multiple events.

For example, when a traveler begins a leg, the persistent bit representing *on leg* (0x4) is set, and the change bit representing *change in on leg* (0x8) is set. While the traveler is on the leg, the persistent bit (0x4) remains set, and the change bit (0x8) is cleared. When the traveler ends the leg, the persistent bit (0x4) is cleared, and the change bit (0x8) is again set for one timestep. While the traveler is not on a leg (e.g., while waiting somewhere) both the persistent bit (0x4) and the change bit (0x8) are cleared.

A few of the status bits occur singly rather than in pairs because both bits are not required. For example, a persistent bit for *on trip* is not needed because travelers are only simulated while they are on a trip. A persistent bit that is always set provides no additional information and clutters the

output, and therefore is not used. The *non-motorized* bit (0x20) is used in conjunction with the *on leg* bits to indicate that the leg does not involve vehicular travel.

The *location type identification* bits (0x4000, 0x8000, and 0x4000000) are used in two ways: They are used in conjunction with bits 0x1000 and 0x2000 to identify the type of the location at which the traveler is waiting. They are also used to specify the type of location when the LOCATION field represents a parking place or transit stop ID. For example, when a traveler begins a leg at a parking place, bit 0x4000 will be set in addition to bits 0x4 and 0x8 to signify that the beginning location of the leg is a parking place.

The DISTANCESUM field accumulates the distance traveled along links and within intersections. Upon entering the intersection, DISTANCESUM is incremented by the setback on the link just left, and when exiting the intersection, DISTANCESUM is incremented by the setback on new link.

8.2.2 Vehicle Snapshot

Vehicle snapshot data provides information about vehicles traveling on a link. When collected for every link on every timestep, this gives a complete *trajectory* for each vehicle in the simulation. Vehicle snapshot data is collected as frequently as the analyst indicates in the input configuration file for the specified links.

Table 64: Vehicle snapshot record fields.

Field	Interpretation
VEHICLE	Vehicle ID.
TIME	Current time (seconds from midnight).
LINK	Link ID on which the vehicle was traveling.
NODE	Node ID vehicle was traveling away from.
LANE	Number of the lane on which the vehicle is traveling.
DISTANCE	Distance (in meters) the vehicle is away from the setback of the node from which it is traveling away .
VELOCITY	Velocity (in meters per second) of the vehicle.
VEHTYPE	Vehicle type: 0 = walk 1 = auto 2 = truck 3 = bicycle 4 = taxi 5 = bus 6 = trolley 7 = streetcar 8 = light rail 9 = rapid rail 10 = regional rail
ACCELER	Acceleration (in meters per second) the vehicle had in the current timestep.
DRIVER	Driver ID.
PASSENGERS	Count of passengers in vehicle.
EASTING	Vehicle's *x*-coordinate (in meters).
NORTHING	Vehicle's *y*-coordinate (in meters).
ELEVATION	Vehicle's *z*-coordinate (in meters).
AZIMUTH	Vehicle's orientation angle (degrees from east in the counterclockwise direction).
USER	User-defined field that can be set on a per-vehicle basis.

8.2.3 Intersection Snapshot

Intersection snapshot data provides information about a vehicle as it is traversing an intersection. This data is collected as frequently as the analyst indicates in the input configuration file for the specified nodes.

Table 65: Intersection snapshot record fields.

Field	Interpretation
VEHICLE	Vehicle ID.
TIME	Current time (seconds from the midnight).
NODE	Node ID where the vehicle is located.
LINK	Link ID from which the vehicle entered.
LANE	Number of the lane from which the vehicle entered.
QINDEX	Vehicle position in the intersection buffer.

8.2.4 Traffic Control Snapshot

Traffic control snapshot data reports the current state of the traffic signal at a node. This data is collected as frequently as the analyst indicates in the input configuration file for the specified nodes.

Table 66: Traffic control snapshot record fields.

Field	Interpretation
NODE	Node ID where the signal is located.
TIME	Current time (seconds from midnight).
LINK	Link ID entering the signal.
LANE	Number of the lane entering the signal.
SIGNAL	Type of control present: 0: None 1: Stop 2: Yield 3: Wait 4: Caution 5: Permitted 6: Protected

8.2.5 Link Travel Times Summary

Link travel time summary data reports counts of vehicles and travel times on links accumulated as vehicles exit the links. This data is collected as frequently as the analyst indicates in the input configuration file for the specified links. For IOC-2, there are separate data records for each turning movement leaving each lane on the link.

Table 67: Link travel times summary field records.

Field	Interpretation
LINK	Link ID being reported.
NODE	Node ID from which the vehicles were traveling away.
TIME	Current time (seconds from midnight).
COUNT	Number of vehicles leaving the link.
SUM	Sum of the vehicle travel times (in seconds) for vehicles leaving the link. (The time spent in the previous intersection is included in this value.)
SUMSQUARES	Sum of the vehicle travel time squares (in seconds squared) for vehicles leaving the link. (The time spent in the previous intersection is included in this value.)
TURN	Type of turn the vehicle made leaving the link.
LANE	Lane number.
VCOUNT	Number of vehicles on the link.
VSUM	Sum of vehicle velocities (in meters per second) on the link.
VSUMSQUARES	Sum of the squares of the vehicle velocities (in meters squared per second squared).

8.2.6 Link Densities Summary

Link density summary data reports counts and velocities of vehicles within *boxes* that partition the link. This data is collected as frequently as the analyst indicates in the input configuration file for the specified links. For IOC-2, there are separate data records for each lane on the link. The box length is specified in the input configuration file.

Table 68: Link densities summary record fields.

Field	Interpretation
LINK	Link ID being reported.
NODE	Node ID from which the vehicles were traveling away.
DISTANCE	Ending distance of the box (in meters) from the setback of the node from which the vehicles were traveling away.
TIME	Current time (seconds from midnight).
COUNT	Number of vehicles in the box.
SUM	Sum of the vehicle velocities (in meters per second) in the box.
SUMSQUARES	Sum of the squares of the vehicle velocities (in meters squared per second squared).
LANE	Lane number.

8.2.7 Link Velocities Summary

Link velocity summary data reports histograms of velocities of vehicles within *boxes* that partition the link. This data is collected as frequently as the analyst indicates in the input configuration file for the specified links. The box length, number of histogram bins, and maximum velocity are specified in the input configuration file. For the microsimulation used in IOC-2, the maximum velocity is typically 37.5 m/s, and the velocity range is divided into five bins plus an overflow bin extending to infinity. Histogram intervals are defined to be closed at the lower end of the bin and open at the upper end.

Table 69: Link velocities summary record fields.

Field	Interpretation
LINK	Link ID being reported.
NODE	Node ID from which the vehicles were traveling away.
DISTANCE	Ending distance of the box (in meters) from the setback of the node from which the vehicles were traveling away.
TIME	Current time (seconds from midnight).
COUNT0	Number of vehicles with velocities in the range [0, 7.5).
COUNT1	Number of vehicles with velocities in the range [7.5, 15).
COUNT2	Number of vehicles with velocities in the range [15, 22.5).
COUNT3	Number of vehicles with velocities in the range [22.5, 30).
COUNT4	Number of vehicles with velocities in the range [30, 37.5).
COUNT5	Number of vehicles with velocities in the range [37.5, infinity).

8.2.8 Link Energy Summary

Link energy summary data reports histograms of energies (integrated power) of vehicles accumulated as vehicles enter the links. Energy is defined as the sum of the vehicle's power over each timestep, where power is defined as the velocity times the acceleration when the acceleration is greater than zero. Vehicles are assumed to have zero power while they are in intersections. The units for energy in IOC-2 are cells-squared per second-squared. (See the documentation for the microsimulation for the definition of a cell.)

☛ Link energy summary data is not used by the Emissions Estimator in this release.

This data is collected as frequently as the analyst indicates in the input configuration file for the specified links. The number of histogram bins and maximum energy is specified in the input configuration file. Histogram intervals are defined to be closed at the lower end of the bin and open at the upper end.

Table 70: Link energy summary record fields.

Field	Interpretation
LINK	Link ID being reported.
NODE	Node ID from which the vehicles were traveling away.
TIME	Current time (seconds from midnight).
ENERGY0	Number of vehicles with integrated power in the range [0, *energy_maximum* / *number_bins*).
ENERGY1	Number of vehicles with integrated power in the second bin.
ENERGY2	Number of vehicles with integrated power in the third bin.
ENERGYn	Number of vehicles with integrated power in the range [energy_maximum, infinity).

8.3 Output Filtering

A variety of output filtering capabilities are provided in order to limit potentially voluminous output to only those items of interest in a particular simulation run. An unlimited number of output specifications may be included in the simulation configuration file, allowing for very fine-grained control of the output that is produced in the input configuration file.

Time-based filtering may be used to restrict data collection to a subset of the total run time by specifying a starting and ending time. The frequency of reporting for evolution and summary data and the sampling frequency for summary data are specified by the analyst in the input configuration file.

Data collected may be restricted to a subset of nodes and links in the road network. Table 71 describes the fields in the node specification file, and Table 72 describes the fields in the link specification files. Regional filtering allows the specification of the corners of a rectangular region in which data should be collected. (Note that the microsimulator does not currently utilize regional filtering.)

Table 71: Node specification fields.

Field	Description
NAME	Output file name.

Field	Description
NODE	Node ID.

Table 72: Link specification fields.

Field	Description
NAME	Output file name.
LINK	Link ID.

Data may be filtered by value, with only those items that pass all filters appearing in the output. The supported operators for value filtering are indicated in Table 73. Data fields in a record may be suppressed, resulting in shorter records.

Table 73: Value filtering operators.

Operators	Interpretation
==	equal to
!=	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
%	an integer multiple of
!%	not an integer multiple of
#	included in the list (a list is a string of values starting with the character [, ending with the character], and where each value is separated by the character |)
!#	not included in the list
&	has set bits
!&	has cleared bits

8.4 Interface Functions

8.4.1 OutReadHeader

Signature: `int OutReadHeader(FILE * file, TOutHeader * header)`

Description: Read a header from an output table.

Argument: `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure defined in Section 8.5.1.

Return Value: Return nonzero if the header was successfully read, or zero if not.

8.4.2 OutWriteHeader

Signature: `int OutWriteHeader (FILE * file, const TOutHeader * header)`

Description: Write a header to an output table.

Argument: `file` – pointer to a `FILE` stream object.

`header` – pointer to an output table header structure defined in Section 8.5.1.

Return Value: Return nonzero if the header was successfully written, or zero if not.

8.4.3 OutSkipHeader

Signature: `int` **`OutSkipHeader`** `(FILE * file)`

Description: Skip a header from an output table.

Argument: `file` – pointer to a `FILE` stream object.

Return Value: Return nonzero if the header was successfully skipped, or zero is not.

8.4.4 OutReadNodeSpecification

Signature: `int` **`OutReadNodeSpecification`** `(FILE * file, TOutNodeSpecificationRecord * record)`

Description: Read a record from a node specification table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to an output node specification record structure defined in Section 8.5.2.

Return Value: Return nonzero if the record was successfully read, or zero if not.

8.4.5 OutWriteNodeSpecification

Signature: `int` **`OutWriteNodeSpecification`** `(FILE * file, const TOutNodeSpecificationRecord * record)`

Description: Write a record to a node specification table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to an output node specification record structure defined in Section 8.5.2.

Return Value: Return nonzero if the record was successfully written, or zero if not.

8.4.6 OutReadLinkSpecification

Signature: `int` **`OutReadLinkSpecification`** `(FILE * file, TOutLinkSpecificationRecord * record)`

Description: Read a record from a link specification table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to an output link specification structure defined in Section 8.5.3.

Return Value: Return nonzero if the record was successfully read, or zero if not.

8.4.7 OutWriteLinkSpecification

Signature: `int **OutWriteLinkSpecification** (FILE * const TOutLinkSpecificationRecord * record)`

Description: Write a record to a link specification table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to an output link specification structure defined in Section 8.5.3.

Return Value: Return nonzero if the record was successfully written, or zero if not.

8.4.8 OutReadTravelerEventHeader

Signature: `int **OutReadTravelerEventHeader** (FILE * file, TOutHeader * header, TOutTravelerEventRecord * record)`

Description: Read a header from a traveler event table.

Argument: `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure defined in Section 8.5.1.
`record` – pointer to a traveler event structure defined in Section 8.5.4.

Return Value: Return nonzero if the header was successfully read, or zero if not.

8.4.9 OutWriteTravelerEventHeader

Signature: `int **OutWriteTravelEventHeader** (FILE * file, const TOutHeader * header, TOutTravelerEventRecord * record)`

Description: Write a header to a traveler event table.

Argument: `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure defined in Section 8.5.1.
`record` – pointer to a traveler event structure defined in Section 8.5.4.

Return Value: Return nonzero if the header was successfully written, or zero if not.

8.4.10 OutReadTravelerEvent

Signature: `int **OutReadTravelerEvent** (FILE * file, TOutTravelerEventRecord * record)`

Description: Read a record from a traveler event table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to a traveler event structure defined in Section 8.5.4.

Return Value: Return nonzero if the record was successfully read, or zero if not.

8.4.11 OutWriteTravelerEvent

Signature:
```
int OutWriteTravelerEvent (FILE * file, const
TOutTravelerEventRecord * record)
```

Description: Write a record to a traveler event table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to a traveler event structure defined in Section 8.5.4.

Return Value: Return nonzero if the record was successfully written, or zero if not.

8.4.12 OutReadVehicleEvolutionHeader

Signature:
```
int OutReadVehicleEvolutionHeader (FILE * file,
TOutHeader * header, TOutVehicleEvolutionRecord * record)
```

Description: Read a header from a vehicle evolution table.

Argument: `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure defined in Section 8.5.1.
`record` – pointer to a vehicle evolution structure defined in Section 8.5.5.

Return Value: Return nonzero if the header was successfully read, or zero if not.

8.4.13 OutWriteVehicleEvolutionHeader

Signature:
```
int OutWriteVehicleEvolutionHeader (FILE * file, const
TOutHeader * header, TOutVehicleEvolutionRecord * record)
```

Description: Write a header to a vehicle evolution table.

Argument: `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure defined in Section 8.5.1.
`record` – pointer to a vehicle evolution structure defined in Section 8.5.5.

Return Value: Return nonzero if the header was successfully written, or zero if not.

8.4.14 OutReadVehicleEvolution

Signature:
```
int OutReadVehicleEvolution (FILE * file,
TOutVehicleEvolutionRecord * record)
```

Description: Read a record from a vehicle evolution table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to a vehicle evolution structure defined in Section 8.5.5.

Return Value: Return nonzero if the record was successfully read, or zero if not.

8.4.15 OutWriteVehicleEvolution

Signature: `int OutWriteVehicleEvolution (FILE * file, const TOutVehicleEvolutionRecord * record)`

Description: Write a record to a vehicle evolution table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to a vehicle evolution structure defined in Section 8.5.5.

Return Value: Return nonzero if the record was successfully written, or zero if not.

8.4.16 OutReadIntersectionEvolutionHeader

Signature: `int OutReadIntersectionEvolutionHeader (FILE * file, TOutHeader * header, TOutIntersectionEvolutionRecord * record)`

Description: Read a header from an intersection evolution table.

Argument: `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure defined in Section 8.5.1.
`record` – pointer to an intersection evolution structure defined in Section 8.5.6.

Return Value: Return nonzero if the header was successfully read, or zero if not.

8.4.17 OutWriteIntersectionEvolutionHeader

Signature: `int OutWriteIntersectionEvolutionHeader (FILE * file, const TOutHeader * header, TOutIntersectionEvolutionRecord * record)`

Description: Write a header to an intersection evolution table.

Argument: `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure defined in Section 8.5.1.
`record` – pointer to an intersection evolution structure defined in Section 8.5.6.

Return Value: Return nonzero if the header was successfully written, or zero if not.

8.4.18 OutReadIntersectionEvolution

Signature: `int OutReadIntersectionEvolution (FILE * file, TOutIntersectionEvolutionRecord * record)`

Description: Read a record from an intersection evolution table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to an intersection evolution structure defined in Section 8.5.6.

Return Value: Return nonzero if the record was successfully read, or zero if not.

8.4.19 OutWriteIntersectionEvolution

Signature: `int` **`OutWriteIntersectionEvolution`** `(FILE * file, const TOutIntersectionEvolutionRecord * record)`

Description: Write a record to an intersection evolution table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to an intersection evolution structure defined in Section 8.5.6.

Return Value: Return nonzero if the record was successfully written, or zero if not.

8.4.20 OutReadTrafficControlEvolutionHeader

Signature: `int` **`OutReadTrafficControlEvolutionHeader`** `(FILE * file, TOutHeader * header, TOutTrafficControlEvolutionRecord * record)`

Description: Read a header from a traffic control evolution table.

Argument: `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure defined in Section 8.5.1.
`record` – pointer to a traffic control evolution structure defined in Section 8.5.7.

Return Value: Return nonzero if the header was successfully read, or zero if not.

8.4.21 OutWriteTrafficControlEvolutionHeader

Signature: `int` **`OutWriteTrafficControlEvolutionHeader`** `(FILE * file, const TOutHeader * header, TOutTrafficControlEvolutionRecord * record)`

Description: Write a header to a traffic control evolution table.

Argument: `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure defined in Section 8.5.1.
`record` – pointer to a traffic control evolution structure defined in Section 8.5.7.

Return Value: Return nonzero if the header was successfully written, or zero if not.

8.4.22 OutReadTrafficControlEvolution

Signature: `int` **`OutReadTrafficControlEvolution`** `(FILE * file, TOutTrafficControlEvolutionRecord * record)`

Description: Read a record from a traffic control evolution table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to a traffic control evolution structure defined in Section 8.5.7.

Return Value: Return nonzero if the record was successfully read, or zero if not.

8.4.23 OutWriteTrafficControlEvolution

Signature: `int` **`OutWriteTrafficControlEvolution`** `(FILE * file, const TOutTrafficControlEvolutionRecord * record)`

Description: Write a record to a traffic control evolution table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to a traffic control evolution structure defined in Section 8.5.7.

Return Value: Return nonzero if the record was successfully written, or zero if not.

8.4.24 OutReadLinkTimeSummaryHeader

Signature: `int` **`OutReadLinkTimeSummaryHeader`** `(FILE * file, TOutHeader * header, TOutLinkTimeSummaryRecord * record)`

Description: Read a header from a link time summary table.

Argument: `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure defined in Section 8.5.1.
`record` – pointer to a link time summary structure defined in Section 8.5.8.

Return Value: Return nonzero if the header was successfully read, or zero if not.

8.4.25 OutWriteLinkTimeSummaryHeader

Signature: `int` **`OutWriteLinkTimeSuummaryHeader`** `(FILE * file, const TOutHeader * header, TOutLinkTimeSummaryRecord * record)`

Description: Write a header to a link time summary table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to an output table header structure defined in Section 8.5.1.
`TOutLinkTimeSummaryRecord *` – pointer to a link time summary structure defined in Section 8.5.8.

Return Value: Return nonzero if the header was successfully written, or zero if not.

8.4.26 OutReadLinkTimeSummary

Signature: `int` **`OutReadLinkTimeSummary`** `(FILE * file,`

```
TOutLinkTimeSummaryRecord * record)
```

Description: Read a record from a link time summary table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to a link time summary structure defined in Section 8.5.8.

Return Value: Return nonzero if the record was successfully read, or zero if not.

8.4.27 OutWriteLinkTimeSummary

Signature:
```
int OutWriteLinkTimeSummary (FILE * file, const
TOutLinkTimeSummaryRecord * record)
```

Description: Write a record to a link time summary table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to a link time summary structure defined in Section 8.5.8.

Return Value: Return nonzero if the record was successfully written, or zero if not.

8.4.28 OutReadLinkSpaceSummaryHeader

Signature:
```
int OutReadLinkSpaceSummaryHeader (FILE * file,
TOutHeader * header, TOutLinkSpaceSummaryRecord *
record)
```

Description: Read a header from a link space summary table.

Argument: `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure defined in Section 8.5.1.
`record` – pointer to a link space summary structure defined in Section 8.5.9.

Return Value: Return nonzero if the header was successfully read, or zero if not.

8.4.29 OutWriteLinkSpaceSummaryHeader

Signature:
```
int OutWriteLinkSpaceSummaryHeader (FILE * file, const
TOutHeader * header, TOutLinkSpaceSummaryRecord * record)
```

Description: Write a header to a link space summary table.

Argument: `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure defined in Section 8.5.1.
`record` – pointer to a link space summary structure defined in Section 8.5.9.

Return Value: Return nonzero if the header was successfully written, or zero if not.

8.4.30 OutReadLinkSpaceSummary

Signature:
```
int OutReadLinkSpaceSummary (FILE * file,
TOutLinkSpaceSummaryRecord * record)
```

Description: Read a record from a link space summary table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to a link space summary structure defined in Section 8.5.9.

Return Value: Return nonzero if the record was successfully read, or zero if not.

8.4.31 OutWriteLinkSpaceSummary

Signature: `int` **`OutWriteLinkSpaceSummary`** `(FILE * file, const TOutLinkSpaceSummaryRecord * record)`

Description: Write a record to a link space summary table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to a link space summary structure defined in Section 8.5.9.

Return Value: Return nonzero if the record was successfully written, or zero if not.

8.4.32 OutReadLinkVelocitySummaryHeader

Signature: `int` **`OutReadLinkVelocitySummaryHeader`** `(FILE * file, TOutHeader * header, TOutLinkVelocitySummaryRecord * record)`

Description: Read a header from a link velocity summary table.

Argument: `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table structure defined in Section 8.5.1.
`TOutLinkVelocitySummaryRecord` – pointer to a link velocity summary structure defined in Section 8.5.10.

Return Value: Return nonzero if the header was successfully read, or zero if not.

8.4.33 OutWriteLinkVelocitySummaryHeader

Signature: `int` **`OutWriteLinkVelocitySummaryHeader`** `(FILE * file, const TOutHeader * head, TOutLinkVelocitySummaryRecord * record)`

Description: Write a header to a link velocity summary table.

Argument: `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure defined in Section 8.5.1.
`record` – pointer to a link velocity summary structure defined in Section 8.5.10.

Return Value: Return nonzero if the header was successfully written, or zero if not.

8.4.34 OutReadLinkVelocitySummary

Signature: `int` **`OutReadLinkVelocitySummary`** `(FILE * file, TOutLinkVelocitySummaryRecord * record)`

Description: Read a record to a link velocity summary table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to a link velocity summary structure defined in Section 8.5.10.

Return Value: Return nonzero if the record was successfully read, or zero if not.

8.4.35 OutWriteLinkVelocitySummary

Signature: `int` **`OutWriteLinkVelocitySummary`** `(FILE * file, const TOutLinkVelocitySummaryRecord * record)`

Description: Write a record to a link velocity summary table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to a link velocity summary structure defined in Section 8.5.10.

Return Value: Return nonzero if the record was successfully written, or zero if not.

8.4.36 OutReadLinkEnergySummaryHeader

Signature: `int` **`OutReadLinkEnergySummaryHeader`** `(FILE * file, TOutHeader * header, TOutLinkEnergySummaryRecord * record)`

Description: Read a header to a link energy summary table.

Argument: `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure defined in Section 8.5.1.
`record` – pointer to a link energy summary structure defined in Section 8.5.11.

Return Value: Return nonzero if the header was successfully read, or zero if not.

8.4.37 OutWriteLinkEnergySummaryHeader

Signature: `int` **`OutWriteLinkEnergySummaryHeader`** `(FILE * file, const TOutHeader *header, TOutLinkEnergySummaryRecord * record)`

Description: Write a header to a link energy summary table.

Argument: `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure defined in

Section 8.5.1.
`record` – pointer to a link energy summary structure defined in Section 8.5.11.

Return Value: Return nonzero if the header was successfully written, or zero if not.

8.4.38 OutReadLinkEnergySummary

Signature: `int OutReadLinkEnergySummary (FILE * file, TOutLinkEnergySummaryRecord * record)`

Description: Read a record to a link energy summary table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to a link energy summary structure defined in Section 8.5.11.

Return Value: Return nonzero if the record was successfully read, or zero if not.

8.4.39 OutWriteLinkEnergySummary

Signature: `int OutWriteLinkEnergySummary (FILE * file, const TOutLinkEnergySummaryRecord * record)`

Description: Write a record to a link energy summary table.

Argument: `file` – pointer to a `FILE` stream object.
`record` – pointer to a link energy summary structure defined in Section 8.5.11.

Return Value: Return nonzero if the record was successfully written, or zero if not.

8.5 Data Structures

8.5.1 TOutHeader

This structure is used for the output table header.

```
typedef struct
{
/** The field names. **/
INT8 fFields[512];

} TOutHeader;
```

8.5.2 TOutNodeSpecificationRecord

This structure is used for output node specification table records.

```
typedef struct
{
/** The NAME field. **/
```

```
INT8 fName[100];

/** The NODE field. **/
INT32 fNode;

} TOutNodeSpecificationRecord;
```

8.5.3 TOutLinkSpecificationRecord

This structure is used for output link specification table records.

```
typedef struct
{
/** The NAME field. **/
INT8 fName[100];

/** The LINK field. **/
INT32 fLink;

} TOutLinkSpecificationRecord;
```

8.5.4 TOutTravelerEventRecord

This structure is used for traveler event records.

```
typedef structure
{
/** The TIME field. **/
REAL64 fTime;

/** The TRAVELER field.  **/
INT32 fTraveler;

/** The TRIP field. **/
INT32 fTrip;

/** The LEG field. **/
INT32 fLeg;

/** The VEHICLE field. **/
INT32 fVehicle;

/** The VEHTYPE field. **/
INT32 fVehtype;

/** The VSUBTYPE field. **/
INT32 fVsubtype;

/** The ROUTE field. **/
INT32 fRoute;

/** The STOPS field. **/
INT32 fStops;

/** The YIELDS field. **/
INT32 fYields;

/** The SIGNALS field. **/
```

```
INT32 fSignals;

/** The TURN field. **/
INT32 fTurn;

/** The STOPPED field. **/
REAL64 fStopped;

/** the ACCELS field. **/
REAL64 fAccels;

/** The TIMESUM field. **/
REAL64 fTimesum;

/** The DISTANCESUM field. **/
REAL64 fDistancesum;

/** The USER field. **/
INT32 fUser;

/** The Anomaly field. **/
INT32 fAnomaly;

/** The STATUS field. **/
INT32 fStatus;

/** The LOCATION field. **/
INT32 fLocation;

/** Private:  The i/o formats. **/
INT8 fFormat[2] [85];

/** Private: The pointers to the data. **/
INT32 fOffsets[20];

} TOutTravelerEventRecord;
```

8.5.5 TOutVehicleEvolutionRecord

This structure is used for vehicle evolution records.

```
typedef struct
{
/** The TIME field. **/
REAL64 fTime;

/** The DRIVER field. **/
INT32 fDriver;

/** The VEHICLE field. **/
INT32 fVehicle;

/** The VEHTYPE field. **/
INT32 fVehtype;

/** The LINK field. **/
INT32 fLink;

/** The NODE field. **/
```

```
INT32 fNode.;

/** The LANE field. **/
INT32 fLane;

/** The DISTANCE field. **/
REAL64 fDistance;

/** The VELOCITY field. **/
REAL64 fVelocity;

/** The ACCELER field. **/
REAL64 fAcceler;

/** The PASSENGERS field. **/
INT32 fPassengers;

/** The EASTING field. **/
REAL64 fEasting;

/** The NORTHING field. **/
REAL64 fNorthing;

/** The ELEVATION field. **/
REAL64 fElevation;

/** The AZIMUTH field. **/
REAL64 fAzimuth;

/** The USER field. **/
INT32 fUser;

/** Private: The i/o formats. **/
INT8 fFormat[2] [72];

/** Private: The pointers to the data. **/
INT32 fOffsets[16];

} TOutVehicleEvolutionRecord;
```

8.5.6 TOutIntersectionEvolutionRecord

This structure is used for intersection evolution records.

```
typedef struct
{
/** The TIME field. **/
REAL64 fTime;

/** The VEHICLE field. **/
INT32 fVehicle;

/** The NODE field. **/
INT32 fNode;

/** The LINK field. **/
INT32 fLink;

/** The LANE field. **/
```

```
INT32 fLane;

/** The QINDEX field. **/
INT32 fQindex;

/** Private: The i/o formats. **/
INT8 fFormat[2] [25];

/** Private:  The pointer to the data. **/
INT32 fOffsets [6];

} TOutIntersectionEvolutionRecord;
```

8.5.7 TOutTrafficControlEvolutionRecord;

This structure is used for traffic control evolution records.

```
typedef struct
{
/** The TIME field. **/
REAL64 fTime;

/** The NODE field. **/
INT32 fNode;

/** The LINK field. **/
INT32 fLink;

/** The LANE field. **/
INT32 fLane;

/** The SIGNAL field. **/
INT32 fSignal;

/** Private: The i/o formats. **/
INT8 fFormat [2] [21];

/** Private: The pointers to the data. **/
INT32 fOffsets[5];

} TOutTrafficControlEvolutionRecord;
```

8.5.8 TOutLinkTimeSummaryRecord

This structure is used for link time summary records.

```
typedef struct
{
/** The TIME field. **/
REAL64 fTime;

/** The LINK field. **/
INT32 fLink;

/** The NODE field. **/
INT32 fNode;

/** The LANE field. **/
```

```
INT32 fLane;

/** The TURN field. **/
INT32 fTurn;

/** The COUNT field. **/
INT32 fCount;

/** The SUM field. **/
REAL64 fSum;

/** The SUMSQUARES field. **/
REAL64 fSumsquares;

/** The VCOUNT field. **/
INT32 fVCount;

/** The VSUM field. **/
REAL64 fVSum;

/** The VSUMSQUARES field. **/
REAL64 fVSumsquares;

/** Private: The i/o formats. **/
INT8 fFormat[2] [49];

/** Private: The pointers to the data. **/
INT32 fOffsets[11];

} TOutLinkTimeSummaryRecord;
```

8.5.9 TOutLinkSpaceSummaryRecord

This structure is used for link space summary records.

```
typedef struct
{
/** The TIME field. **/
REAL64 fTime;

/** The LINK field. **/
INT32 fLink;

/** The NODE field. **/
INT32 fNode;

/** The LANE field. **/
INT32 fLane;

/** The DISTANCE field. **/
REAL64 fDistance;

/** The COUNT field. **/
INT32 fCount;

/** The SUM field. **/
REAL64 fSum;

/** The SUMSQUARES field. **/
```

```
REAL64 fSumsquares;

/** Private: The i/o formats. **/
INT8 fFormat[2] [36];

/** Private: The pointers to the data. **/
INT32 fOffsets[8];

} TOutLinkSpaceSummaryRecord;
```

8.5.10 TOutLinkVelocitySummaryRecord

This structure is used for link velocity summary records.

```
/** Maximum allowed number of bins in a histogram. **/
#define HISTOGRAM_MAX_BINS 100

/** Structure for link velocity summary records. **/
typedef struct
{

/** The TIME field. **/
REAL64 fTime;

/** The LINK field. **/
INT32 fLink;

/** The NODE field. **/
INT32 fNode.

/** The DISTANCE field. **/
READ64 fDistance;

/** The COUNT fields. **/
INT32 fCount [HISTOGRAM_MAX_BINS];

/** The number of bins in the histogram. **/
INT32 fNumberBins;

/** Private: The i/o formats. **/
INT8 fFormat[2] [18 + 4 * HISTOGRAM_MAX_BINS];

/** Private:  The pointers to the data. **/
INT32 fOffsets[4 + HISTOGRAM_MAX_BINS];

} TOutLinkVelocitySummaryRecord;
```

8.5.11 TOutLinkEnergySummaryRecord

This structure is used for link energy summary records.

```
/** Maximum allowed number of bins in a histogram. **/
#define HISTOGRAM_MAX_BINS_100

/** Structure for link energy summary records. **/
typedef struct
{
/** The TIME field. **/
```

```
REAL64 fTime;

/** The LINK field. **/
INT32 fLink;

/** The NODE field. **/
INT32 fNode;

/** The ENERGY fields. **/
INT32 fEnergy[HISTOGRAM_MAX_BINS];

/** The number of bins in the histogram. **/
INT32 fNumberBins;

/** Private: The i/o formats. **/
INT8 fFormat[2] [13 + 3 * HISTOGRAM_MAX_BINS];

/** Private: The pointers to the data. **/
INT32 fOffsets[3 + HISTOGRAM_MAX_BINS];


} TOutLinkEnergySummaryRecord;
```

8.6 Utility Programs

8.6.1 InterpretStatus

InterpretStatus displays the STATUS field in the traveler event output data as a bit pattern for easier interpretation.

Usage: `InterpretStatus <event file>`

InterpretStatus reads the event file and writes the bit patterns representing the STATUS field to standard output. The output may be redirected to a file if preferred.

8.6.2 TestSimOutput

TestSimOutput tests much of the functionality of the simulation output representation. Its primary use is for regression testing when the output representation is modified.

Usage: `TestSimOutput <configuration file>`

TestSimOutput writes output to standard out. The final line should be "No failures occurred."

☛ *TestSimOutput* is not available in this release.

8.6.3 CompareDensity

CompareDensity allows vehicle evolution data and link density summary data to be compared for verification of consistency. Comparison of new output with previously recorded output allows a limited form of regression testing of simulation output when the simulation is modified.

Usage: `CompareDensity <configuration file>`

CompareDensity writes records that are not within the tolerated difference to standard output.

☛ *CompareDensity* is not available in this release.

8.6.4 CompareVelocity

CompareVelocity allows vehicle evolution data and link velocity summary data to be compared for verification of consistency. Comparison of new output with previously recorded output allows a limited form of regression testing of simulation output when the simulation is modified.

Usage: `CompareVelocity <configuration file>`

CompareVelocity writes records that are not within the tolerated difference to standard output.

☛ *CompareVelocity* is not available in this release.

8.6.5 DumpOutput

DumpOutput may be used to merge and filter binary output collected on individual computational nodes and convert it to ASCII format. Binary data collection is still available in the output representation, but has been largely superceded by the ASCII data collection capability provided by the parallel toolbox. *DumpOutput* is still available, but is now seldom used.

☛ *DumpOutput* is not available in this release.

8.6.6 SetupOutput

The *SetupOutput* script copies a set of empty and test output tables into a specified directory. It takes the name of the directory as its only argument.

8.6.7 CleanupOutput

The *CleanupOutput* script removes a set of tables created by *SetupOutput*. It takes the name of the directory as its argument.

8.7 Files

Table 74: Simulation output library files.

Type	File Name	Description
Binary Files	libTIO.a	TRANSIMS Interfaces library
Source Files	outio.c	Defines simulation output data structures and interface functions
	outio.h	Simulation output interface functions source file
Utilities	InterpretStatus	Interprets event status field
	SetupOutput	Creates empty and test output files
	CleanupOutput	Removes empty and test output files
Example Files	Test*.tbl	Tests output tables
	TestConfiguration.tbl	Configuration file for *TestSimOutput*

8.8 Configuration Keys

In the simulation output keywords, the trailing *n* must be replaced by an integer, beginning with 1 for the first set of output of each type (snapshot, event, and summary). If more than one set of output is desired for a particular type, the second set of keywords ends with *n*=2; the third set uses *n*=3, etc. There is no restriction to the number of output data sets of each type that may be requested.

The keywords in Table 75 pertain to the snapshot (evolution) type of output.

Table 75: Configuration keys for snapshot output.

Key	Description
OUT_SNAPSHOT_NAME_n	file name for snapshot output
OUT_SNAPSHOT_TYPE_n	types of snapshot output to collect (separated by semicolons) permissible values are VEHICLE; INTERSECTION; SIGNAL
OUT_SNAPSHOT_BEGIN_TIME_n	first time (in seconds from the midnight before simulation start) at which to collect data
OUT_SNAPSHOT_END_TIME_n	last time (in seconds from the midnight before simulation start) at which to collect data
OUT_SNAPSHOT_TIME_STEP_n	frequency (in seconds) at which to report data (i.e., write it to disk)
OUT_SNAPSHOT_EASTING_MIN_n	minimum easting (in meters) for which to report data (currently unused)
OUT_SNAPSHOT_EASTING_MAX_n	maximum easting (in meters) for which to report data (currently unused)
OUT_SNAPSHOT_NORTHING_MIN_n	minimum northing (in meters) for which to report data (currently unused)
OUT_SNAPSHOT_NORTHING_MAX_n	maximum northing (in meters) for which to report data (currently unused)
OUT_SNAPSHOT_NODES_n	path of the node specification file (file is described in Table 71)
OUT_SNAPSHOT_LINKS_n	path of the link specification file (file is described in Table 72)
OUT_SNAPSHOT_SUPPRESS_n	list of fields (separated by semicolons) not to include in the output file
OUT_SNAPSHOT_FILTER_n	list of expressions (of the form FIELD OPERATOR VALUE, and separated by semicolons) for filtering records; (valid values for FIELD are found in Table 64 through Table 66, and values for OPERATOR are found in Table 73)

The keywords in Table 76 pertain to the event type of output.

Table 76: Configuration keys for event output.

Key	Description
OUT_EVENT_NAME_n	file name for event output
OUT_EVENT_TYPE_n	types of event output to collect permissible value is TRAVELER
OUT_EVENT_BEGIN_TIME_n	first time (in seconds from the midnight before simulation start) at which to collect data
OUT_EVENT_END_TIME_n	last time (in seconds from the midnight before simulation start) at which to collect data
OUT_EVENT_EASTING_MIN_n	minimum easting (in meters) for which to report data (currently unused)
OUT_EVENT_EASTING_MAX_n	maximum easting (in meters) for which to report data (currently unused)
OUT_EVENT_NORTHING_MIN_n	minimum northing (in meters) for which to report data (currently unused)
OUT_EVENT_NORTHING_MAX_n	maximum northing (in meters) for which to report data (currently unused)
OUT_EVENT_SUPPRESS_n	list of fields (separated by semicolons) not to include in the output file
OUT_EVENT_FILTER_n	list of expressions (of the form FIELDNAME OPERATOR VALUE, and separated by semicolons) for filtering records; (valid values for FIELD are found in Table 62 and valid values for OPERATOR are found in Table 73)

The keywords in Table 77 pertain to the summary type of output.

Table 77: Configuration keys for summary output.

Key	Description
OUT_SUMMARY_NAME_n	file name for summary output
OUT_SUMMARY_TYPE_n	types of summary output to collect (separated by semicolons) permissible values are DENSITY; TIME; VELOCITY; ENERGY
OUT_SUMMARY_BEGIN_TIME_n	first time (in seconds from the midnight before simulation start) at which to collect data
OUT_SUMMARY_END_TIME_n	last time (in seconds from the midnight before simulation start) at which to collect data
OUT_SUMMARY_TIME_STEP_n	frequency (in seconds) at which to report data (i.e., write it to disk)
OUT_SUMMARY_SAMPLE_TIME_n	frequency (in seconds) at which to accumulate data
OUT_SUMMARY_BOX_LENGTH_n	length of the boxes (in meters)
OUT_SUMMARY_EASTING_MIN_n	minimum easting (in meters) for which to report data (currently unused)
OUT_SUMMARY_EASTING_MAX_n	maximum easting (in meters) for which to report data (currently unused)
OUT_SUMMARY_NORTHING_MIN_n	minimum northing (in meters) for which to report data (currently unused)
OUT_SUMMARY_NORTHING_MAX_n	maximum northing (in meters) for which to report data (currently unused)
OUT_SUMMARY_LINKS_n	path of the link specification file (file is described in Table 72)
OUT_SUMMARY_SUPPRESS_n	list of fields (separated by semicolons) not to include in the output file
OUT_SUMMARY_FILTER_n	list of expressions (of the form FIELDNAME OPERATOR VALUE, and separated by semicolons) for filtering records; (valid values for FIELD are found in Table 67 and Table 68, and valid values for OPERATOR are found in Table 73)
OUT_SUMMARY_VELOCITY_BINS_n	number of bins used to cover the range of the velocity histogram
OUT_SUMMARY_VELOCITY_MAX_n	maximum velocity in the velocity histogram
OUT_SUMMARY_ENERGY_BINS_n	number of bins used to cover the range of the energy histogram
OUT_SUMMARY_ENERGY_MAX_n	maximum energy in the energy histogram

The keywords in Table 78 are used only by the *CompareDensity* and *CompareVelocity* programs. Only the first of these keywords is used by *CompareVelocity*.

Table 78: Configuration keys for the *CompareDensity* and *CompareVelocity* programs.

Key	Description
OUT_SUMMARY_SPACE_COUNT_TOLERANCE_1	difference tolerated between snapshot and summary count data
OUT_SUMMARY_SPACE_SUM_TOLERANCE_1	difference tolerated between snapshot and summary sum data
OUT_SUMMARY_SPACE_SUMSQUARES_TOLERANCE_1	difference tolerated between snapshot and summary sum-of-squares data

8.9 Examples

The example presented in this section uses the example network presented in Section 7.8. Table 79 presents a small set of plans that are simulated on the network.

Table 79: Plan set.

Trip/Leg	Plan	Description
Traveler 101, Trip 1, Leg 1	101 3 1 1 1 0 24600 1002 2 1003 2 400 24600 1 1 0 1 6 300 0 8520 14141 8522 8521	Traveler 10 drives auto 300 from parking 1002 to parking 1003 via nodes 8520, 14141, 8522, 8521.
Traveler 101, Trip 1, Leg 2	101 3 1 2 0 0 2500 1003 2 3002 3 120 25000 1 0 2 0 0	Traveler 101 walks from parking 1003 to transit stop 3002.
Traveler 1, Trip 1, Leg 1	1 10 1 1 1 0 25200 1005 2 1006 2 300 25200 1 1 1 5 6 100 20 8525 8603 14340 8608	Traveler 1 drives bus 100 along bus route 20 from parking 1005 to parking 1006 via nodes 8525, 8603, 14340, 8608.
Traveler 101, Trip 1, Leg 3	101 3 1 3 0 0 25200 3002 3 3005 3 300 25300 1 0 1 5 1 20	Traveler 101 rides bus from transit stop 3002 to transit stop 3005 along bus route 20.
Traveler 1, Trip 1, Leg 2	1 10 1 2 0 0 25500 1006 2 1006 2 0 25800 1 0 4 0 0	Traveler 1 has a layover activity at parking 1006 from the time of arrival until time 25800 seconds past midnight.
Traveler 101, Trip 1, Leg 4	101 3 1 4 0 1 25500 3005 3 1006 2 30 25500 1 0 2 0 0	Traveler 101 walks from transit stop 3005 to parking 1006.
Traveler 1, Trip 1, Leg 3	1 10 1 3 0 1 25800 1006 2 1005 2 200 25800 1 1 1 5 6 100 21 8608 14340 8603 8525	Traveler 1 drives bus 100 along bus route 21 from parking 1006 to parking 1005 via nodes 8608, 14340, 8603, 8525.

The following is an excerpt of the simulation configuration file that pertains to the output collected.

```
# directory for simulation output (all output for a simulation is written to a
# single directory)
OUT_DIRECTORY                        /home/Gershwinoutput1/kpb4jh

# file name for snapshot output
OUT_SNAPSHOT_NAME_1                  output.test.evol

# first time (in seconds from the midnight before simulation start) at which to
# collect data
OUT_SNAPSHOT_BEGIN_TIME_1            24610

# last time (in seconds from the midnight before simulation start) at which to
# collect data
OUT_SNAPSHOT_END_TIME_1              86400

# frequency (in seconds) at which to report data (i.e., write it to disk)
OUT_SNAPSHOT_TIME_STEP_1             1

# path of the node specification file
OUT_SNAPSHOT_NODES_1                 /home/projects/transims/database/test/Test_Out
                                     put_Node_Specification_Table

# path of the link specification file
OUT_SNAPSHOT_LINKS_1                 /home/projects/transims/database/test/Test_Out
                                     put_Link_Specification_Table

# file name for event output
OUT_EVENT_NAME_1                     output.test.event

# first time (in seconds from the midnight before simulation start) at which to
# collect data
OUT_EVENT_BEGIN_TIME_1               0

# last time (in seconds from the midnight before simulation start) at which to
# collect data
OUT_EVENT_END_TIME_1                 86400

# file name for event output
OUT_SUMMARY_NAME_1                   output.test.sum

# first time (in seconds from the midnight before simulation start) at which to
# collect data
OUT_SUMMARY_BEGIN_TIME_1             24610

# last time (in seconds from the midnight before simulation start) at which to
# collect data
OUT_SUMMARY_END_TIME_1               86400

# frequency (in seconds) at which to report data (i.e., write it to disk)
OUT_SUMMARY_TIME_STEP_1              900

# frequency (in seconds) at which to accumulate data
OUT_SUMMARY_SAMPLE_TIME_1            60

# length of the boxes (in meters)
OUT_SUMMARY_BOX_LENGTH_1             150

# path of the link specification file (file is described in Table 56)
OUT_SUMMARY_LINKS_1                  /home/projects/transims/database/test/Test_Out
                                     put_Link_Specification_Table
```

Table 80 (parts a and b) shows the traveler event output that was collected for an 1800-second simulation.

Table 80a: Traveler event output.

	ACCELS	ANOMALY	DISTANCESUM	LEG	LOCATION	ROUTE	SIGNALS	STATUS	STOPPED	STOPS
A	0	0	0	1	1002	-1	0	16412	0	0
B	0	0	307.5	1	1002	-1	0	17156	0	0
C	0	0	307.5	1	1002	-1	0	19716	0	0
D	0	0	135	1	12384	-1	0	16778501	0	0
E	0	0	694	1	12384	-1	0	1286	0	0
F	0	0	2194	1	28800	-1	1	1286	59	0
G	0	0	3194	1	11487	-1	1	1286	59	0
H	0	0	5694	1	9705	-1	2	1286	63	0
I	0	0	6499.5	1	12407	-1	2	1286	63	0
J	0	0	6499.5	1	1003	-1	2	18692	63	0
K	0	0	6499.5	1	1003	-1	2	16900	63	0
L	0	0	6499.5	1	1003	-1	2	16392	63	0
M	0	0	0	2	1003	-1	0	16428	0	0
N	0	0	0	2	3002	20	0	32808	0	0
O	0	0	0	3	3002	20	0	32780	0	0
P	0	0	0	3	3002	20	0	45060	0	0
Q	0	0	0	1	1005	20	0	16412	0	0
R	0	0	0	1	1005	20	0	28676	0	0
S	0	0	0	1	1005	20	0	21252	0	0
T	0	0	0	1	1005	20	0	23812	0	0
U	0	0	0	1	1005	20	0	25860	0	0
V	1	0	15	1	3002	20	0	230661	0	0
W	0	0	7.5	3	3002	20	0	37636	0	0
X	0	0	7.5	3	3002	20	0	41220	0	0
Y	1	0	37.5	1	3002	20	0	132357	0	0
Z	2	0	374.5	1	2758	20	0	1286	0	0
AA	1	0	367	3	2758	20	0	262	0	0
BB	2	0	1374.5	1	2759	20	0	1286	0	0
CC	1	0	1367	3	2759	20	0	262	0	0
DD	21	0	4874.5	1	2750	20	0	1286	1	0
EE	20	0	4867	3	2750	20	0	262	1	0
FF	21	0	5874.5	1	2751	20	0	1286	1	1
GG	20	0	5867	3	2751	20	0	262	1	1
HH	21	0	6203	1	3005	20	0	230661	1	1
II	20	0	6195.5	3	3005	20	0	33284	1	1
JJ	20	0	6195.5	3	3005	-1	0	32776	1	1
KK	0	0	0	4	3005	-1	0	32812	0	0
LL	21	0	6233	1	3005	20	0	132357	1	1
MM	21	0	6233	1	2752	20	0	1286	1	1
NN	21	0	6225.5	1	1006	20	0	18692	1	1
OO	21	0	6225.5	1	1006	20	0	16900	1	1
PP	21	0	6225.5	1	1006	-1	0	16392	1	1
QQ	0	0	0	2	1006	-1	0	16428	0	0
RR	0	0	0	2	1006	-1	0	802852	0	0
SS	0	0	0	2	1006	21	0	540708	0	0
TT	0	0	0	2	1006	21	0	16424	0	0
UU	0	0	0	3	1006	21	0	16396	0	0
VV	0	0	0	3	1006	21	0	28676	0	0
WW	0	0	0	3	1006	21	0	21252	0	0
XX	0	0	0	3	1006	21	0	23812	0	0
YY	0	0	0	3	1006	21	0	25860	0	0
ZZ	1	0	353.5	3	2752	21	0	1286	0	0
AAA	1	0	1353.5	3	2751	21	0	1286	2	0
BBB	1	0	4853.5	3	2750	21	0	1286	3	1
CCC	1	0	5853.5	3	2759	21	0	1286	4	1
DDD	1	0	6225.5	3	2758	21	0	1286	4	1
EEE	1	0	6495.5	3	1005	21	0	18692	4	1
FFF	1	0	6495.5	3	1005	21	0	16900	4	1
GGG	1	0	6495.5	3	1005	21	0	16408	4	1

Table 80b: Traveler output data.

	TIME	TIMESUM	TRAVELER	TRIP	TURN	USER	VEHICLE	VEHTYPE	VSUBTYPE	YIELDS
A	24610	10	101	1	0	3	0	0	0	0
B	24610	10	101	1	0	3	300	1	0	0
C	24610	10	101	1	0	3	300	1	0	0
D	24610	0	101	1	0	3	300	1	0	0
E	24638	28	101	1	0	3	300	1	0	0
F	24780	170	101	1	0	3	300	1	0	1
G	24827	217	101	1	-1	3	300	1	0	1
H	24947	337	101	1	-1	3	300	1	0	1
I	24986	376	101	1	-1	3	300	1	0	1
J	24986	376	101	1	-1	3	300	1	0	1
K	24986	376	101	1	-1	3	300	1	0	1
L	24986	376	101	1	-1	3	0	0	0	1
M	24986	0	101	1	0	3	0	0	0	0
N	25106	0	101	1	0	3	0	0	0	0
O	25106	0	101	1	0	3	0	0	0	0
P	25106	0	101	1	0	3	0	0	0	0
Q	25200	0	1	1	0	10	0	0	0	0
R	25201	0	1	1	0	10	0	0	0	0
S	25201	0	1	1	0	10	100	5	0	0
T	25201	0	1	1	0	10	100	5	0	0
U	25201	0	1	1	0	10	100	5	0	0
V	25203	2	1	1	0	10	100	5	0	0
W	25209	103	101	1	0	3	100	5	0	0
X	25209	103	101	1	0	3	100	5	0	0
Y	25209	8	1	1	0	10	100	5	0	0
Z	25231	30	1	1	0	10	100	5	0	0
AA	25231	125	101	1	0	3	100	5	0	0
BB	25304	103	1	1	0	10	100	5	0	0
CC	25304	198	101	1	0	3	100	5	0	0
DD	25612	411	1	1	0	10	100	5	0	0
EE	25612	506	101	1	0	3	100	5	0	0
FF	25684	483	1	1	0	10	100	5	0	0
GG	25684	578	101	1	0	3	100	5	0	0
HH	25708	507	1	1	0	10	100	5	0	0
II	25712	606	101	1	0	3	100	5	0	0
JJ	25712	606	101	1	0	3	0	0	0	0
KK	25712	0	101	1	0	3	0	0	0	0
LL	25712	511	1	1	0	10	100	5	0	0
MM	25712	511	1	1	0	10	100	5	0	0
NN	25712	511	1	1	0	10	100	5	0	0
OO	25712	511	1	1	0	10	100	5	0	0
PP	25712	511	1	1	0	10	0	0	0	0
QQ	25712	0	1	1	0	10	0	0	0	0
RR	25712	0	1	1	0	10	0	0	0	0
SS	25712	0	1	1	0	10	0	0	0	0
TT	25712	0	1	1	0	10	0	0	0	0
UU	25712	0	1	1	0	10	0	0	0	0
VV	25800	0	1	1	0	10	0	0	0	0
WW	25800	0	1	1	0	10	100	5	0	0
XX	25800	0	1	1	0	10	100	5	0	0
YY	25800	0	1	1	0	10	100	5	0	0
ZZ	25823	23	1	1	0	10	100	5	0	0
AAA	25897	97	1	1	0	10	100	5	0	0
BBB	26153	353	1	1	0	10	100	5	0	0
CCC	26225	425	1	1	0	10	100	5	0	0
DDD	26250	450	1	1	0	10	100	5	0	0
EEE	26250	450	1	1	0	10	100	5	0	0
FFF	26250	450	1	1	0	10	100	5	0	0
GGG	26250	450	1	1	0	10	0	0	0	0

Table 81 (parts a and b) shows the first 30 seconds of vehicle snapshot data collected.

Table 81a: First 30 seconds of vehicle snapshot data.

	AZIMUTH	DISTANCE	DRIVER	EASTING	ELEVATION	LANE	LINK
A	90	442.5	101	3005.25	1000	2	12384
B	90	465	101	3005.25	1000	2	12384
C	90	480	101	3005.25	1000	2	12384
D	90	502.5	101	3005.25	1000	2	12384
E	90	517.5	101	3005.25	999.99994	2	12384
F	90	540	101	3005.25	1000	2	12384
G	90	562.5	101	3005.25	1000.0001	2	12384
H	90	577.5	101	3005.25	1000	2	12384
I	90	600	101	3005.25	1000	2	12384
J	90	622.5	101	3005.25	1000	2	12384
K	90	645	101	3005.25	1000	2	12384
L	90	667.5	101	3005.25	1000	2	12384
M	90	690	101	3005.25	1000	2	12384
N	90	712.5	101	3005.25	1000	2	12384
O	90	735	101	3005.25	1000	2	12384
P	90	750	101	3005.25	1000	2	12384
Q	90	765	101	3005.25	1000	2	12384
R	90	787.5	101	3005.25	1000	2	12384
S	90	802.5	101	3005.25	1000	2	12384
T	90	825	101	3005.25	1000	2	12384
U	90	847.5	101	3005.25	1000	2	12384
V	90	862.5	101	3005.25	1000	2	12384
W	90	885	101	3005.25	1000.0001	2	12384
X	90	900	101	3005.25	1000	2	12384
Y	90	922.5	101	3005.25	1000	2	12384
Z	90	937.5	101	3005.25	1000	2	12384
AA	90	960	101	3005.25	1000	2	12384
BB	90	982.5	101	3005.25	1000	2	12384
CC	90	15	101	3005.25	1000	2	28800
DD	90	30	101	3005.25	1000	2	28800
EE	90	52.5	101	3005.25	1000	2	28800

Table 81b: First 30 seconds of vehicle snapshot data.

	NODE	NORTHING	PASSENGERS	TIME	VEHICLE	VEHTYPE	VELOCITY
A	14136	1948.5	0	24610	300	1	15
B	14136	1971	0	24611	300	1	22.5
C	14136	1986	0	24612	300	1	15
D	14136	2008.5	0	24613	300	1	22.5
E	14136	2023.4999	0	24614	300	1	15
F	14136	2046	0	24615	300	1	22.5
G	14136	2068.5	0	24616	300	1	22.5
H	14136	2083.5	0	24617	300	1	15
I	14136	2106	0	24618	300	1	22.5
J	14136	2128.5	0	24619	300	1	22.5
K	14136	2151	0	24620	300	1	22.5
L	14136	2173.5	0	24621	300	1	22.5
M	14136	2196	0	24622	300	1	22.5
N	14136	2218.5	0	24623	300	1	22.5
O	14136	2241	0	24624	300	1	22.5
P	14136	2256	0	24625	300	1	15
Q	14136	2271	0	24626	300	1	15
R	14136	2293.5	0	24627	300	1	22.5
S	14136	2308.5	0	24628	300	1	15
T	14136	2331	0	24629	300	1	22.5
U	14136	2353.5	0	24630	300	1	22.5
V	14136	2368.5	0	24631	300	1	15
W	14136	2391	0	24632	300	1	22.5
X	14136	2406	0	24633	300	1	15
Y	14136	2428.5	0	24634	300	1	22.5
Z	14136	2443.5	0	24635	300	1	15
AA	14136	2466	0	24636	300	1	22.5
BB	14136	2488.5	0	24637	300	1	22.5
CC	8520	2515	0	24638	300	1	22.5
DD	8520	2530	0	24639	300	1	15
EE	8520	2552.5	0	24640	300	1	22.5

Table 82 shows the intersection snapshot data collected during the entire simulation.

Table 82: Intersection snapshot data.

LANE	LINK	NODE	QINDEX	TIME	VEHICLE
2	28800	14141	1	24780	300
1	9705	8521	1	24947	300

Table 83 shows the signal snapshot data collected during the first second of the simulation.

Table 83: Signal snapshot data.

LANE	LINK	NODE	SIGNAL	TIME
1	9704	8521	5	24610
2	9704	8521	5	24610
1	9705	8521	3	24610
1	12407	8521	5	24610
2	12407	8521	5	24610
3	12407	8521	3	24610
1	9706	8521	3	24610
1	11487	14141	3	24610
2	11487	14141	3	24610
3	11487	14141	3	24610
4	11487	14141	3	24610
5	11487	14141	3	24610
6	11487	14141	6	24610
1	11486	14141	5	24610
2	11486	14141	5	24610
3	11486	14141	5	24610
1	11495	14141	3	24610
2	11495	14141	3	24610
3	11495	14141	3	24610
4	11495	14141	3	24610
5	11495	14141	3	24610
6	11495	14141	7	24610
1	28800	14141	3	24610
2	28800	14141	3	24610
3	28800	14141	5	24610
4	28800	14141	5	24610
5	28800	14141	5	24610
6	28800	14141	6	24610

Table 84 shows the travel time summary data collected every 15 minutes.

Table 84: Travel time summary data.

COUNT	LANE	LINK	NODE	SUM	SUMSQUARES	TIME	TURN	VCOUNT	VSUM	VSUMSQUARES
1	1	11487	14141	47	2209	25510	-1	0	0	0
1	1	9705	8522	120	14400	25510	-1	0	0	0
1	2	28800	8520	142	20164	25510	-1	0	0	0
1	2	2759	8525	73	5329	25510	0	0	0	0
0	2	2759	8525	0	0	26410	0	0	0	0
1	1	2759	8603	72	5184	26410	0	0	0	0
1	2	2751	14340	72	5184	26410	0	0	0	0
1	1	2751	8608	74	5476	26410	0	0	0	0
0	1	11487	14141	0	0	26410	-1	0	0	0
1	2	2750	8603	308	94864	26410	0	0	0	0
1	1	2750	14340	256	65536	26410	0	0	0	0
0	1	9705	8522	0	0	26410	-1	0	0	0
0	2	28800	8520	0	0	26410	-1	0	0	0

Table 85 shows the link density summary table that was collected for one link at the first summary collection time.

Table 85: Link density summary table.

COUNT	DISTANCE	LANE	LINK	NODE	SUM	SUMSQUARES	TIME
0	975	1	2759	8525	0	0	25510
0	975	2	2759	8525	0	0	25510
0	975	3	2759	8525	0	0	25510
0	825	1	2759	8525	0	0	25510
0	825	2	2759	8525	0	0	25510
0	825	3	2759	8525	0	0	25510
0	675	1	2759	8525	0	0	25510
0	675	2	2759	8525	0	0	25510
0	675	3	2759	8525	0	0	25510
0	525	1	2759	8525	0	0	25510
1	525	2	2759	8525	7.5	56.25	25510
0	525	3	2759	8525	0	0	25510
0	375	1	2759	8525	0	0	25510
0	375	2	2759	8525	0	0	25510
0	375	3	2759	8525	0	0	25510
0	225	1	2759	8525	0	0	25510
0	225	2	2759	8525	0	0	25510
0	225	3	2759	8525	0	0	25510
0	75	1	2759	8525	0	0	25510
0	75	2	2759	8525	0	0	25510
0	75	3	2759	8525	0	0	25510
0	975	1	2759	8603	0	0	25510
0	975	2	2759	8603	0	0	25510
0	825	1	2759	8603	0	0	25510
0	825	2	2759	8603	0	0	25510
0	675	1	2759	8603	0	0	25510
0	675	2	2759	8603	0	0	25510
0	525	1	2759	8603	0	0	25510
0	525	2	2759	8603	0	0	25510
0	375	1	2759	8603	0	0	25510
0	375	2	2759	8603	0	0	25510
0	225	1	2759	8603	0	0	25510
0	225	2	2759	8603	0	0	25510
0	75	1	2759	8603	0	0	25510
0	75	2	2759	8603	0	0	25510

Table 86 shows the link velocity summary data that were collected for one link at the first summary collection time.

Table 86: Link velocity summary data.

COUNT0	COUNT1	COUNT2	COUNT3	COUNT4	COUNT5	DISTANCE	LINK	NODE	TIME
0	0	0	0	0	0	150	2759	8525	25510
0	0	0	0	0	0	300	2759	8525	25510
0	0	0	0	0	0	450	2759	8525	25510
0	0	1	0	0	0	600	2759	8525	25510
0	0	0	0	0	0	750	2759	8525	25510
0	0	0	0	0	0	900	2759	8525	25510
0	0	0	0	0	0	975	2759	8525	25510
0	0	0	0	0	0	150	2759	8603	25510
0	0	0	0	0	0	300	2759	8603	25510
0	0	0	0	0	0	450	2759	8603	25510
0	0	0	0	0	0	600	2759	8603	25510
0	0	0	0	0	0	750	2759	8603	25510
0	0	0	0	0	0	900	2759	8603	25510
0	0	0	0	0	0	975	2759	8603	25510

Table 87 shows the link energy summary data that were collected at the first summary collection time.

Table 87: Link energy summary data.

ENERGY0	ENERGY1	ENERGY10	ENERGY11	ENERGY12	ENERGY13	ENERGY14	ENERGY2	ENERGY3	ENERGY4	ENERGY5	ENERGY6	ENERGY7	ENERGY8	ENERGY9	LINK	NODE	TIME
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2757	8606	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2757	8524	25510
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2758	8524	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2758	8525	25510
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2759	8525	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2759	8603	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9704	8521	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9704	8523	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9706	8521	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9706	8524	25510
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12384	14136	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12384	8520	25510
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12407	8521	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12407	14136	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	28804	14136	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	28804	8525	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2751	14340	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2751	8608	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2752	8608	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2752	14142	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2753	14142	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2753	8610	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2754	8600	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2754	8522	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2755	8610	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2755	8600	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11486	14141	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11486	14142	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11487	8522	25510
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11487	14141	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11495	14141	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11495	14340	25510
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2750	8603	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2750	14340	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9705	8521	25510
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9705	8522	25510
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	28800	8520	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	28800	14141	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2756	8600	25510
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2756	8606	25510

9. EMISSIONS ESTIMATOR

The TRANSIMS Emission Estimator module is designed to calculate emissions in 30-meter segments along a link for chosen time periods (normally 15 minutes). It gives estimates of tailpipe emissions of Nitrogen Oxides (NO_x), Carbon Monoxide (CO), and hydrocarbons from light-duty vehicles. It also gives fuel-consumption that can be used to calculate emissions of Carbon Dioxide (CO_2).

9.1 Terms

Link	A portion of a highway or street with lanes going in a single direction between intersections.
Light-duty Vehicles	Cars, sport-utility vehicles, and small trucks.
Nitrogen-Oxides	Nitric Oxide and Nitrogen Dioxides.
Soak Time	The length of time an engine has been off before the current trip began.
Vehicle Flux	The product of the density of vehicles by their speeds

9.2 File Format

This section describes the file formats of each of the five input files for the Light-Duty Tailpipe Vehicle submodule and the two output files that it produces. The file names are defined in the code but may be changed using the emissions configuration keys. See Volume 2—*Modules* for an explanation of those keys.

9.2.1 readca.out

The file *Readca.out* contains the link velocity summary data produced by the microsimulation described in Section 8.2.7 and reformatted for input into the Emissions Estimator.. The transformation may be performed by using the *readca* program described in Section 9.3. This file is an input file for the Light-Duty Tailpipe submodule. The first five items described in Table 88 (NV through LENGTH) appear in a single record, followed by NV records containing the six COUNT fields in order in each record. This sequence is repeated for each LINK, NODE, and TIME step in the original file.

Table 88: Link velocity fields in *readca.out* (assuming the microsimulation was run with OUT_SUMMARY_VELOCITY_BINS set to 6).

Field	Interpretation
NV	Number of velocity records for this link, equivalent to the number of boxes that partition the link.
TIME	Current time (seconds from midnight).
LINK	Link ID being reported.
NODE	Node ID from which the vehicles were traveling away.
LENGTH	Length of box.
COUNT0	Number of vehicles with velocities in the range [0, 7.5).
COUNT1	Number of vehicles with velocities in the range [7.5, 15).
COUNT2	Number of vehicles with velocities in the range [15, 22.5).
COUNT3	Number of vehicles with velocities in the range [22.5, 30).
COUNT4	Number of vehicles with velocities in the range [30, 37.5).
COUNT5	Number of vehicles with velocities in the range [37.5, infinity).

9.2.2 ARRAY.INP

The file *ARRAY.INP* is used in conjunction with *array.out* and contains parameters describing the number of records and increments used in *array.out*. Several fields are unused by the Light-Duty Tailpipe submodule.

Table 89: Fields in *ARRAY.INP*.

Field	Interpretation
T0	Time since engine start; not used.
RGRADE0	Representative minimum grade; not used.
DRGRADE	Spacing in grade arrays; presently not used.
V0ARRAY	Representative speed for the lowest speed index (mph); not used.
DVARRAY	Speed bin size (mph).
A0ARRAY	Acceleration for lowest acceleration index (feet/sec).
DACCARAY	Acceleration bin size (feet/sec).
NGRADE	Number of grades in the emission arrays; not used.
NVARRY	Number of velocity bins in the emission arrays.
NAARRAY	Number of acceleration bins in the emission arrays.

9.2.3 array.out

The file *array.out* gives the composite vehicle emissions in 2-mph speed bins and 1.5 feet/second acceleration bins. This file is an input file for the Light-Duty Tailpipe submodule. The data in this file is for the case when there is no grade in the roadway. The first two lines of the file contain header information that is ignored. Only the data fields are described in Table 90.

Table 90: Composite vehicle emissions fields.

Field	Interpretation
VARRAY	Representative speed (mph) for emissions calculation; not used.
ACARRAY	Representative acceleration for emissions calculation; not used.
HCTIJK	Hydrocarbon tailpipe emission rate (grams/sec).
COTIJK	Carbon monoxide tailpipe emission rate (grams/sec).
NOXTIJK	Nitrogen oxides tailpipe emission rate (grams/sec).
FECON	Fuel consumption rate (grams/sec).

9.2.4 wcemratios

The file *wcemratios* is an input for the Light-Duty Tailpipe submodule that contains ratios of cold emissions to hot engine emissions. It contains eight records, one for each of seven groupings based on integrated velocity-acceleration product, and an additional grouping for engines that have been fully warmed-up. The first grouping has a soak time of 60 minutes, and the groupings appear in order from lowest integrated velocity-acceleration product to highest. The values are a multiplier that represents the ratio of emissions for vehicles beginning a link in the group to the emissions of a vehicle with the same driving pattern and a fully warmed up engine and catalyst.

Table 91: Fields in *wcemratios*.

Field	Interpretation
HCR	Multiplier for hydrocarbon emissions.
COR	Multiplier for carbon monoxide emissions.
XNOXR	Multiplier for nitrogen oxides emissions.
FCR	Multiplier for fuel consumption.

9.2.5 vehcold.dis

The file *vehcold.dis* is an input file for the Light-Duty Tailpipe submodule that is used in conjunction with *wcemratios*. It contains the distribution of vehicles entering the link stratified by the time integrated, velocity-acceleration product and by the time the engine was idle before the start of the current trip. Note that negative accelerations are ignored in the calculation of the time-integrated, velocity-acceleration products. This distribution is used to determine what cold/warm emission ratios should be used.

Table 92: Fields in *vehcold.dis*.

Field	Interpretation
VCOLD1	Fraction of the vehicles entering the link that have had time-integrated, velocity-acceleration products in the range of 0-18 cells squared per second squared after being idle for an hour or more; 18 cells squared per second cubed is the typical amount for a vehicle to accelerate to speed on an arterial from a stop; a cell is 7.5 meters.
VCOLD2	Fraction of the vehicles entering the link that have had time-integrated, velocity-acceleration products in the range of 19-36 cells squared per second squared after being idle for an hour or more.
VCOLD3	Fraction of the vehicles entering the link that have had time-integrated, velocity-acceleration products in the range of 37-54 cells squared per second squared after being idle for an hour or more.
VCOLD4	Fraction of the vehicles entering the link that have had time-integrated, velocity-acceleration products in the range of 55-72 cells squared per second squared after being idle for an hour or more.
VCOLD5	Fraction of the vehicles entering the link that have had time-integrated, velocity-acceleration products in the range of 73-90 cells squared per second squared after being idle for an hour or more.
VCOLD6	Fraction of the vehicles entering the link that have had time-integrated, velocity-acceleration products in the range of 91-108 cells squared per second squared after being idle for an hour or more.
VCOLD7	Fraction of the vehicles entering the link that have had time-integrated, velocity-acceleration products in the range of 109-126 cells squared per second squared after being idle for an hour or more.
VCOLD8	Fraction of the vehicles entering the link that have had time-integrated, velocity-acceleration products greater than 126 cells squared per second squared after being idle for an hour or more or were idle for less than 1 hour.

9.2.6 readart.out

The file *readart.out* is an output file produced by the Light-Duty Tailpipe submodule. It is a debugging file that provides intermediate output for the emission calculations.

The first record contains ICX and DELTAF. The second record contains six values of F. The third record contains six values of DEN. The fourth record contains six values of FIJ. The fifth record contains six values of HIJ. The sixth record contains six values of VEHFLUX and VEHFT. The seventh record contains six values of VEHD and VEHDT. The eighth record contains VBAR, SDEVRAT, VLOWRI, VUPPRI and V2SDEV. The ninth record contains five values of VEHFLUXL. The tenth record contains five values of VEHFLUXM. The eleventh record contains five values of VEHFLUXH. The twelfth through fourteenth records contain five values of SPDBAR and SPDC. The fifteenth through seventeenth records contain eighteen values of PIJ. Records 1 - 17 are repeated NV times.

The final records contain N, XNOSUL, XNOSUC, XNOSUH, COSUL, COSUC, COSUH, V2SDEV, SDEV, PL, PCC, and PH. The final records are repeated NV times.

Table 93: Fields in *readart.out*

Field	Interpretation
ICX	Segment of which the calculations are made.
DELTAF	Width of the highest speed bin, always 24.6 feet per second.
F	Average number of vehicles in a 7.5 meter cell.
DEN	Fitted average number of vehicles per 7.5 meter cell.
FIJ	Estimated average vehicle densities per spatial cell (24.6 feet) and per speed cell (24.6 feet per second).
HIJ	Gradient in estimated average vehicle density in units of number per spatial cell squared per speed cell.
VEHFLUX	Estimated vehicle flux in each speed bin for speed bins 0-5 in units of number times feet per second.
VEHFT	Total estimated vehicle flux.
VEHD	Estimated number of vehicles in each speed bin in each cell
VEHDT	Estimated total number of vehicles in a spatial cell.
VBAR	Estimated mean speed in feet per second.
SDEVRAT	Estimated ratio of the standard-deviation of speed to mean speed.
VLOWRI	Cutoff speed for the slowest one-third of the vehicles defined by flux in feet per second.
VUPPRI	Cutoff speed for the slowest two-thirds of the vehicles defined by flux in feet per second.
V2SDEV	Product of the square of the mean speed and the difference between the speed standard deviation and its low congestion reference value in units of feet cubed per second cubed.
VEHFLUXL	Estimated vehicle flux for the slowest third of the vehicles for the current segment, followed by the that of the next four segments down the link.
VEHFLUXM	Estimated vehicle flux for the middle third of the vehicles for the current segment, followed by the that of the next four segments down the link.
VEHFLUXH	Estimated vehicle flux for the fastest third of the vehicles for the current segment, followed by the that of the next four segments down the link.
SPDBAR	Estimated average cube of the speed in units of feet cubed per second cubed for the current segment followed by that of four following segments down the link.
SPDC	Estimated gradient in the cube of the speed normalized by the cube of a spatial cell per second (24.6**3) in units of inverse feet.
PIJ	First three values give the probability of a hard acceleration for the slowest third, the middle third, and the fastest third of the vehicles for the segment, while the 7th through the 9th give the probability for insignificant accelerations for the slowest, middle, and fastest thirds respectively. Currently, hard decelerations are not included, they would occupy the 13th through 15th slots.
N	Segment for which the output is reported.
XNOSUL	Estimated NO_x emissions for the slowest third in units of grams per 7.5 meter cell.
XNOSUC	Estimated NO_x emissions for the middle third in units of grams per 7.5 meter cell.
XNOSUH	Estimated NO_x emissions for the fastest third in units of grams per 7.5 meter cell.
COSUL	Estimated CO emissions for the slowest third in units of grams per 7.5 meter cell.
COSUC	Estimated CO emissions for the middle third in units of grams per 7.5 meter cell.
COSUH	Estimated CO emissions for the fastest third in units of grams per 7.5 meter cell.
V2SDEV	Product of the square of the mean speed and the difference between the speed standard deviation and its low congestion reference value in units of feet cubed per second cubed.
SDEV	Standard deviation of speed derived from the estimated distribution.
PL	Probability of a hard acceleration in the slowest third; unlike the earlier reference this includes an adjustment if the slowest one-third is in the first speed bin.
PCC	Probability of a hard acceleration in the middle third; unlike the earlier reference this includes an adjustment if the middle one-third is in the first speed bin.
PH	Probability of a hard acceleration in the fastest third; unlike the earlier reference this includes an adjustment if the fastest one-third is in the first speed bin.

9.2.7 emissions.out

The file *emissions.out* is an output file produced by the Light-Duty Tailpipe submodule. This file is written using the variable size box format and is ready to be visualized with the Output Visualizer. Each record contains the five fields required by this format plus six data values as described in Table 94.

Table 94: Emissions output for Output Visualizer.

Field	Interpretation
TIME	Current time (seconds from midnight).
LINK	Link ID being reported.
NODE	Node ID vehicles were traveling away from.
DISTANCE	Ending distance of the box (in meters) from the setback of the node from which the vehicles were traveling away.
LENGTH	Length of box.
VTT	Average speed in feet per second.
NOX	Nitrogen oxides emissions (milligrams per 30 meter segment).
CO	Carbon monoxide emissions (grams per 30 meter segment).
HC	Hydrocarbon emissions (milligrams per 30 meter segment).
FE	Fuel consumption (grams per 30 meter segment).
FLUX	Vehicle flux in number times speed in feet per second.

9.3 Utility Programs

9.3.1 Readca

The *Readca* program transforms the link velocity summary output described in Section 8.2.7 into the format required by the emissions module as described in Section 9.2.1. The link is partitioned into boxes of a constant size except that the last box on the link may be shorter than the others. The *Readca* program proportionally inflates the values for the last box to what might be expected if the box were full sized.

Note that *Readca* includes some assumptions that are more restrictive than the generality in the output available from the microsimulation. The program assumes that the boxes that partition the link are 30 meters long; a value other than 30 for the microsimulation parameter OUT_SUMMARY_BOX_LENGTH used when collecting velocity data will result in velocity summary data that cannot be correctly processed by *Readca*. The *Readca* program assumes exactly six velocity histogram bins are defined as described in Table 88. The simulation needs to be run with the configuration key OUT_SUMMARY_VELOCITY_BINS set to 6 in order for this to be accomplished. The program also assumes that the maximum length of a link is 3600 meters. An error message is produced for links that exceed this length.

9.4 Files

Table 95: Emission Estimator library files.

Type	File Name	Description
Binary Files	libTIO.a	TRANSIMS Interfaces library
Source Files	libGlobals.a	TRANSIMS Global library
	carlastcold.C	Main emissions module that takes microsimulation velocity summary data and outputs emissions that can be displayed in the Output Visualizer
	ENVConfigKeys.h	Defines emissions configuration keys
	readca.C	Reads in a microsimulation velocity summary output file and outputs the velocity data in a format that can be inputted to the main emissions module

9.5 Examples

The examples presented in this section use the calibration 2 network which is the intersection calibration network. Figure 4presents an example of some of the configuration parameters that pertain to the Emissions Estimator.

Figure 4: Example configuration parameters.

```
PLAN_FILE                                   $TRANSIMS_ROOT/data/plans/Tee.plans
VEHICLE_FILE                                $TRANSIMS_ROOT/data/vehicles/Tee.vehicles

OUT_DIRECTORY                               $TRANSIMS_ROOT/output
OUT_SUMMARY_NAME_1                          tee.sum
OUT_SUMMARY_LINKS_1                         $TRANSIMS_ROOT/output-specs/tee_output_links
OUT_SUMMARY_BOX_LENGTH_1                    30
OUT_SUMMARY_TYPE_1                          VELOCITY
OUT_SUMMARY_SAMPLE_TIME_1                   1
OUT_SUMMARY_TIME_STEP_1                     900
OUT_SUMMARY_VELOCITY_BINS_1                 6
OUT_SUMMARY_VELOCITY_MAX_1                  45
OUT_SUMMARY_ENERGY_BINS_1                   14
OUT_SUMMARY_ENERGY_MAX_1                    224

NET_DIRECTORY                               $TRANSIMS_ROOT/data/networks/
NET_NODE_TABLE                              Calibration_2_Nodes
NET_LINK_TABLE                              Calibration_2_Links
NET_POCKET_LANE_TABLE                       Calibration_2_Pocket_Lanes
NET_PARKING_TABLE                           Calibration_2_Parking
NET_LANE_CONNECTIVITY_TABLE                 Calibration_2_Lane_Connectivity
NET_UNSIGNALIZED_NODE_TABLE                 Calibration_2_Unsignalized_Nodes
NET_SIGNALIZED_NODE_TABLE                   Calibration_2_Signalized_Nodes
NET_PHASING_PLAN_TABLE                      Calibration_2_Phasing_Plans
NET_TIMING_PLAN_TABLE                       Calibration_2_Timing_Plans
NET_STUDY_AREA_LINKS_TABLE                  Calibration_2_Study_Links

EMISSIONS_ARRAY_PARAMETERS_FILE             $TRANSIMS_ROOT/data/emissions/ARRAY.INP
EMISSIONS_COMPOSITE_INPUT_FILE              $TRANSIMS_ROOT/data/emissions/array.out
EMISSIONS_MICROSIM_VELOCITY_FILE            readca.out
EMISSIONS_VEHICLE_COLD_DISTRIBUTION         $TRANSIMS_ROOT/data/emissions/vehcold.dis
EMISSIONS_WCEM_RATIOS_FILE                  $TRANSIMS_ROOT/data/emissions/wcemratios
```

A portion of a microsimulation velocity summary file is shown in Figure 5. This data was collected on the intersection calibration network using the configuration parameters set to the values in Figure 4. The data consists of the velocity bins for link 1 starting at node 6 at time step 900. There are seventeen boxes on that particular link. Notice that the last box is only 15 meters long instead of 30 meters..

Figure 5: Example velocity summary file.

COUNT0	COUNT1	COUNT2	COUNT3	COUNT4	COUNT5	COUNT6	DISTANCE		LINK	NODE	TIME
0	0	0	0	0	0	0	30	1	6	900	
0	0	0	0	0	0	0	60	1	6	900	
0	0	0	0	0	0	0	90	1	6	900	
0	0	0	0	0	0	0	120	1	6	900	
0	0	0	0	0	0	0	150	1	6	900	
0	0	0	0	0	0	0	180	1	6	900	
0	0	0	0	0	0	0	210	1	6	900	
0	0	0	0	0	0	0	240	1	6	900	
1334	990	237	157	0	0	0	270	1	6	900	
1383	1983	465	68	232	0	0	300	1	6	900	
880	1714	607	123	36	0	0	330	1	6	900	
406	1035	604	246	76	0	0	360	1	6	900	
199	538	482	356	194	0	0	390	1	6	900	
83	227	296	370	336	0	0	420	1	6	900	
17	60	148	336	508	0	0	450	1	6	900	
7	18	81	307	601	0	0	480	1	6	900	
0	6	45	183	293	0	0	495	1	6	900	

A portion of a *readca.out* file is shown in Figure 6. The *readca.out* file is created by the *Readca* program, which reformats the microsimulation output into a format that can be read in by the Emissions Estimator. Figure 6 contains the output from the sample data in Figure 5.

Figure 6: Example *readca.out* file.

```
 nv=    17   900.0         1       6    15.0
  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00
  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00
  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00
  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00
  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00
  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00
  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00
  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00
  1.3340E+03  9.9000E+02  2.3700E+02  1.5700E+02  0.0000E+00  0.0000E+00
  1.3830E+03  1.9830E+03  4.6500E+02  6.8000E+01  2.3200E+02  0.0000E+00
  8.8000E+02  1.7140E+03  6.0700E+02  1.2300E+02  3.6000E+01  0.0000E+00
  4.0600E+02  1.0350E+03  6.0400E+02  2.4600E+02  7.6000E+01  0.0000E+00
  1.9900E+02  5.3800E+02  4.8200E+02  3.5600E+02  1.9400E+02  0.0000E+00
  8.3000E+01  2.2700E+02  2.9600E+02  3.7000E+02  3.3600E+02  0.0000E+00
  1.7000E+01  6.0000E+01  1.4800E+02  3.3600E+02  5.0800E+02  0.0000E+00
  7.0000E+00  1.8000E+01  8.1000E+01  3.0700E+02  6.0100E+02  0.0000E+00
  0.0000E+00  1.2000E+01  9.0000E+01  3.6600E+02  5.8600E+02  0.0000E+00
```

Figure 7 shows the contents of the *ARRAY.INP* file that is used as input by the Emissions Estimator. It contains the parameters describing the number of records and increments used in the array.out file.

Figure 7: Example *ARRAY.INP* file.

```
 600. -8.00 1.0  0.04 2.   -12. 1.5
 17 40 17
```

Figure 8 shows a portion of the contents of the *array.out* file that is used as input the Emissions Estimator. The *array.out* file contains the composite vehicle emissions in 2-mph speed bins and 1.5 feet/second acceleration bins. In this version of the Emissions Estimator, there are 17 acceleration bins and 40 velocity bins. Figure 8 contains data for all of the velocity bins for the first acceleration bin.

Figure 8: Example *array.out* file.

```
 array for no grade Riverside region composite
      v          acc          hc          co          nox         fuel
     1.0000     -8.1816      0.0064      0.0424      0.0010      0.4867
     3.0000     -8.1816      0.0166      0.0424      0.0010      0.4867
     5.0000     -8.1816      0.0273      0.0424      0.0011      0.4867
     7.0000     -8.1816      0.0405      0.0424      0.0011      0.4867
     9.0000     -8.1816      0.0605      0.0424      0.0015      0.4867
    11.0000     -8.1816      0.0799      0.0424      0.0016      0.4867
    13.0000     -8.1816      0.0998      0.0424      0.0016      0.4867
    15.0000     -8.1816      0.1199      0.0424      0.0016      0.4867
    17.0000     -8.1816      0.1399      0.0424      0.0015      0.4867
    19.0000     -8.1816      0.1592      0.0424      0.0016      0.4867
    21.0000     -8.1816      0.1792      0.0424      0.0016      0.4867
    23.0000     -8.1816      0.1993      0.0424      0.0016      0.4867
    25.0000     -8.1816      0.2192      0.0424      0.0016      0.4867
    27.0000     -8.1816      0.2386      0.0424      0.0016      0.4867
    29.0000     -8.1816      0.2587      0.0424      0.0016      0.4867
    31.0000     -8.1816      0.2787      0.0424      0.0016      0.4867
    33.0000     -8.1816      0.2981      0.0424      0.0016      0.4867
    35.0000     -8.1816      0.3181      0.0424      0.0016      0.4867
    37.0000     -8.1816      0.3380      0.0424      0.0016      0.4867
    39.0000     -8.1816      0.3580      0.0424      0.0016      0.4867
    41.0000     -8.1816      0.3774      0.0424      0.0016      0.4867
    43.0000     -8.1816      0.3974      0.0424      0.0016      0.4867
    45.0000     -8.1816      0.4174      0.0424      0.0016      0.4867
    47.0000     -8.1816      0.4368      0.0424      0.0016      0.4867
    49.0000     -8.1816      0.4568      0.0424      0.0016      0.4867
    51.0000     -8.1816      0.4768      0.0424      0.0016      0.4867
    53.0000     -8.1816      0.4968      0.0424      0.0016      0.4867
    55.0000     -8.1816      0.5163      0.0424      0.0016      0.4867
    57.0000     -8.1816      0.5362      0.0424      0.0016      0.4867
    59.0000     -8.1816      0.5562      0.0424      0.0016      0.4867
    61.0000     -8.1816      0.5756      0.0424      0.0016      0.4867
    63.0000     -8.1816      0.5956      0.0424      0.0016      0.4867
    65.0000     -8.1816      0.6156      0.0424      0.0016      0.4867
    67.0000     -8.1816      0.6356      0.0424      0.0016      0.4867
    69.0000     -8.1816      0.6550      0.0424      0.0016      0.4867
    71.0000     -8.1816      0.6750      0.0424      0.0016      0.4867
    73.0000     -8.1816      0.6950      0.0424      0.0016      0.4867
    75.0000     -8.1816      0.7150      0.0424      0.0016      0.4867
    77.0000     -8.1816      0.7344      0.0424      0.0016      0.4867
    79.0000     -8.1816      0.7544      0.0424      0.0016      0.4867
```

Figure 9shows the contents of the *wcemratios* file that is used as input by the Emissions Estimator. It contains the ratios of cold emissions to hot engine emissions.

Figure 9: Example *wcemratios* file.

```
    2.675115    2.116624    1.499590    1.086956
    1.736057    1.388528    1.166301    1.089943
    1.499989    1.172226    1.069732    1.099898
    1.364250    1.040638    1.021857    1.078881
    1.248882    0.946900    1.022250    1.071876
    1.159666    0.981162    1.084750    1.079329
    1.120635    1.061412    1.119071    1.074729
    1.000000    1.000000    1.000000    1.000000
```

Figure 10 shows the contents of the *vehcold.dis* file that is inputted into the Emissions Estimator. It contains the distribution of vehicles entering the link stratified by the time integrated, velocity-acceleration product and by the time the engine was idle before the start of the current trip.

Figure 10: Example *vehcold.dis* file.

```
0.00
0.00
0.00
0.00
0.00
0.00
0.00
1.00
```

Figure 11 shows a portion of the contents of a *readart.out* file that is created by the Emissions Estimator. The *readart.out* file is a debugging file used to provide immediate output for the emissions calculations. Figure 11 contains the output for calculations done on the link seen in Figure 5and Figure 6 for the first box that actually contains vehicle velocities.

Figure 11: Example readart.out file.

```
icx=    9  deltaf=  24.6
   1334.0    990.0    237.0    157.0      0.0      0.0
   1334.0    990.0    237.0    157.0      0.0      0.0
   5.511E-01   4.090E-01   9.791E-02   6.486E-02   0.000E+00   0.000E+00
   1.632E-02  -2.788E-02   2.586E-03  -5.273E-03   0.000E+00  -0.000E+00



   4.982E+02   5.238E+03   2.994E+03   2.736E+03   0.000E+00   0.000E+00   1.147e+04
   3.335E+02   2.475E+02   5.925E+01   3.925E+01   0.000E+00   0.000E+00   6.795e+02
       16.87      1.22417        25.38        55.29      3973.63
  3821.896  8269.800  7104.944  6757.693  6864.691
  3821.911  8269.743  7104.894  6757.607  6864.621
  3821.843  8269.757  7104.881  6757.621  6864.668
     2474.     3957.     4909.    12280.    34195.      0.0415
    58176.    49331.    45331.   105579.   275911.      0.9770
   299058.   588877.   257405.   469191.   744376.      5.0222
 0.028 1.000 1.000 0.000 0.000 0.000
 0.972 0.000 0.000 0.000 0.000 0.000
 0.000 0.000 0.000 0.000 0.000 0.000
 9.     3.48     2.44     2.23     72.5     53.7     56.2   3973.6     20.7 0.028 1.000
1.000
```

Figure 12 shows a portion of the contents of an *emissions.out* file that is created by the Emissions Estimator. The *emissions.out* file is used as input into the Output Visualizer. Figure 12 contains the data for timestep 900 link 1 running from node 6 as seen in the above examples.

Figure 12: Example *emissions.out* file.

```
TIME    LINK    NODE    DISTANCE        LENGTH  VTT     NOX     CO      HC      FE      FLUX
900     1       6        30.0   30.0     0.0     0.0     0.0     0.0     0.0       0.0
900     1       6        60.0   30.0     0.0     0.0     0.0     0.0     0.0       0.0
900     1       6        90.0   30.0     0.0     0.0     0.0     0.0     0.0       0.0
900     1       6       120.0   30.0     0.0     0.0     0.0     0.0     0.0       0.0
900     1       6       150.0   30.0     0.0     0.0     0.0     0.0     0.0       0.0
900     1       6       180.0   30.0     0.0     0.0     0.0     0.0     0.0       0.0
900     1       6       210.0   30.0     0.0     0.0     0.0     0.0     0.0       0.0
900     1       6       240.0   30.0     0.0     0.0     0.0     0.0     0.0       0.0
900     1       6       270.0   30.0    11.5    32604.5 729.9   77223.1 2960.7  7797.0
900     1       6       300.0   30.0    16.3    12839.2 335.7   34413.7 2908.9  16868.0
900     1       6       330.0   30.0    17.3    21346.9 505.3   59210.4 2921.2  14493.0
900     1       6       360.0   30.0    23.3    80315.2 1580.5  161250.3        5311.7  13772.0
900     1       6       390.0   30.0    31.7    67776.4 1351.8  140743.9        4397.7  14001.0
900     1       6       420.0   30.0    41.4    51067.9 990.5   102631.5        3367.0  13593.0
900     1       6       450.0   30.0    51.9    43343.4 827.7   84388.2 2894.5  13868.0
900     1       6       480.0   30.0    56.2    41092.1 760.3   76519.0 2772.0  14246.0
900     1       6       495.0   15.0    56.2    7165.3  123.3   10860.7 892.5   14809.0
```

Figure 13 shows the Output Visualizer emissions colormaps that were used in this version of the Emissions Estimator to color the network's boxes. Thresholds and their colors are defined in the colormaps. See the section on Visualization for interpretation of this file.

Figure 13: Example Output Visualizer emissions colormaps.

```
6 0.0 80.0 Emissions Velocity Map
1.0 8
20.0 0
40.0 9
60.0 5
80.0 3
6 0.0 70000.0 Emissions Nitrogen Oxide Map
200.0 8
20000.0 0
30000.0 9
50000.0 5
70000.0 3
6 0.0 1200.0 Emissions Carbon Monixide Map
1.0 8
300.0 0
600.0 9
900.0 5
1200.0 3
6 0.0 140000.0 Emissions Hydrocarbons Map
200.0 8
20000.0 0
60000.0 9
100000.0 5
140000.0 3
6 0.0 4000.0 Emissions Fuel Economy Map
20.0 8
1000.0 0
2000.0 9
3000.0 5
4000.0 3
6 0.0 14000.0 Emissions Flux Map
200.0 8
2000.0 0
6000.0 9
10000.0 5
14000.0 3
```

Figure 14 through Figure 19 show examples of visualization of emissions calculated for the Intersection calibration network.

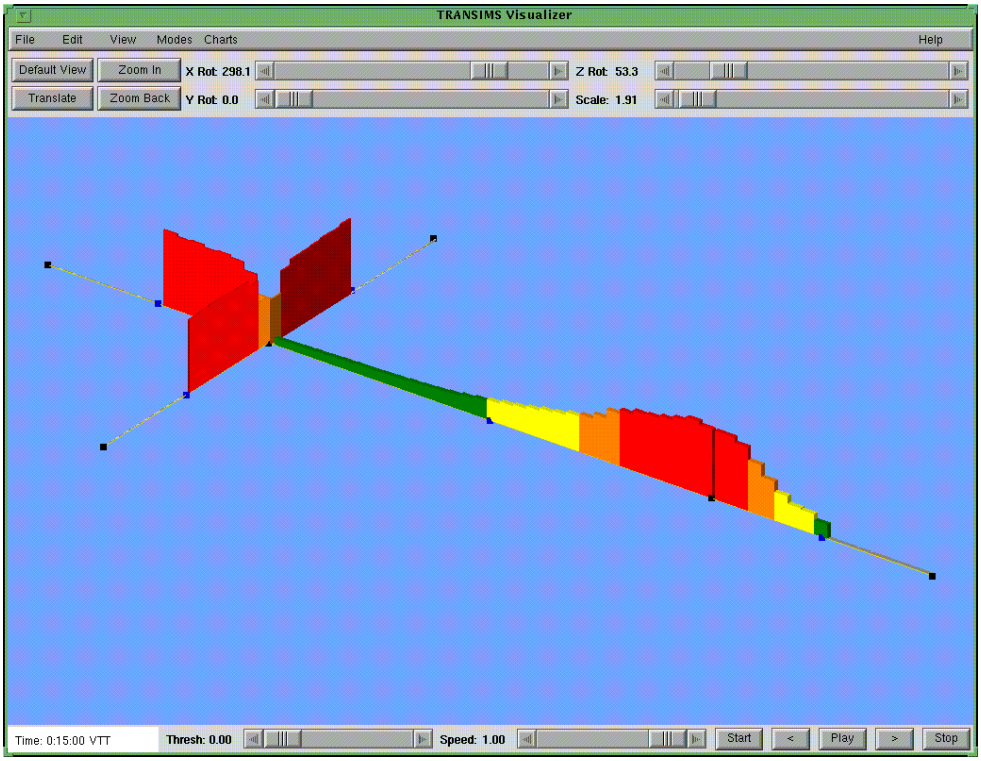

Figure 14: Velocities.

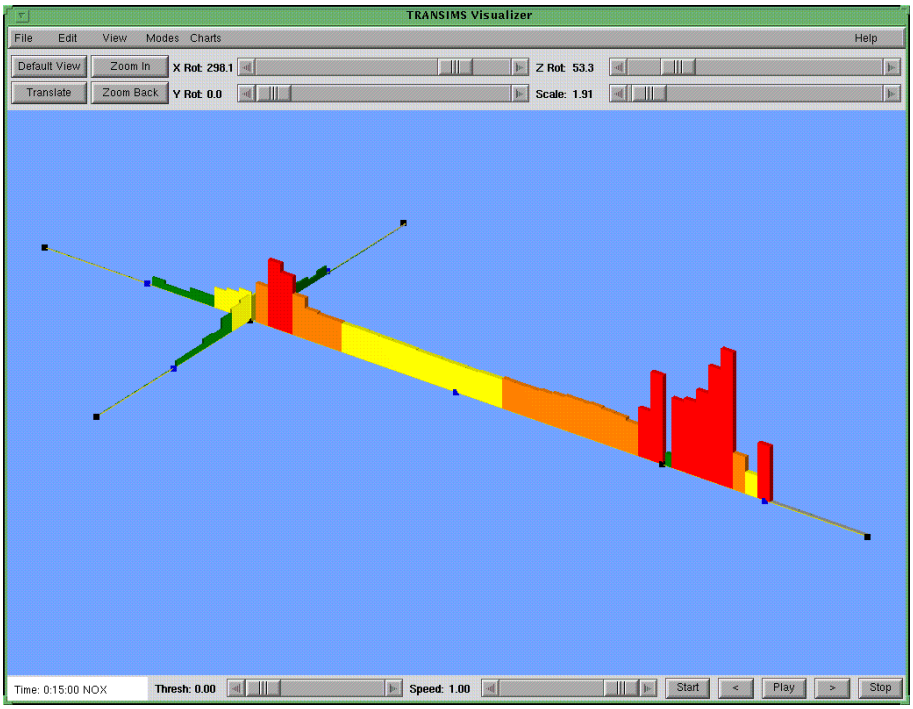

Figure 15: NO_x (nitrogen oxides) emissions.

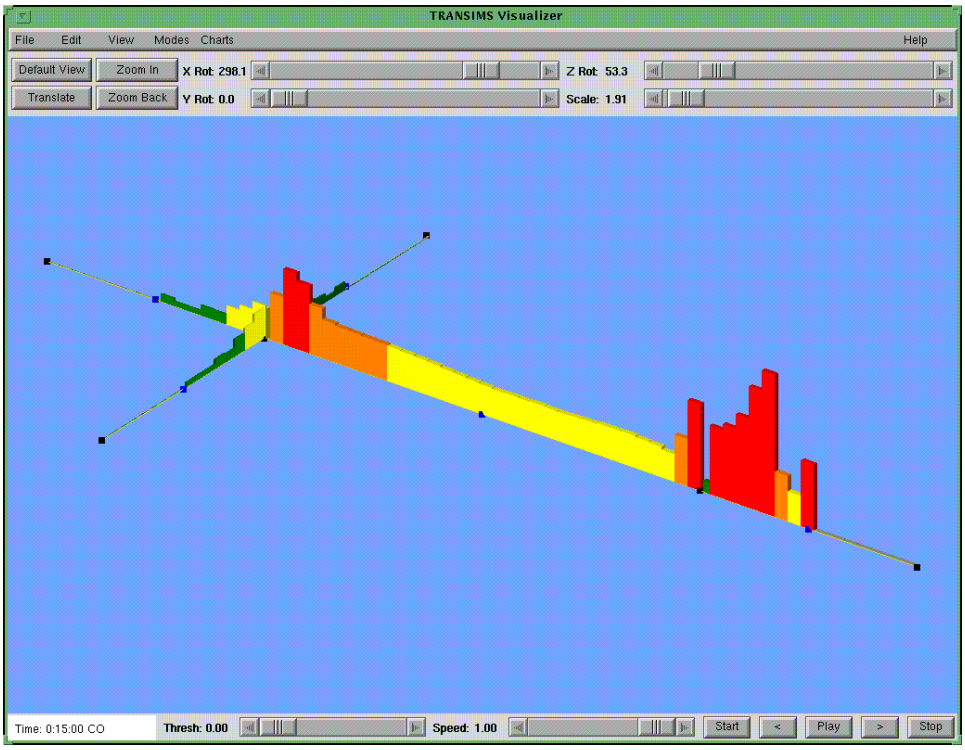

Figure 16: CO (carbon monoxide) emissions.

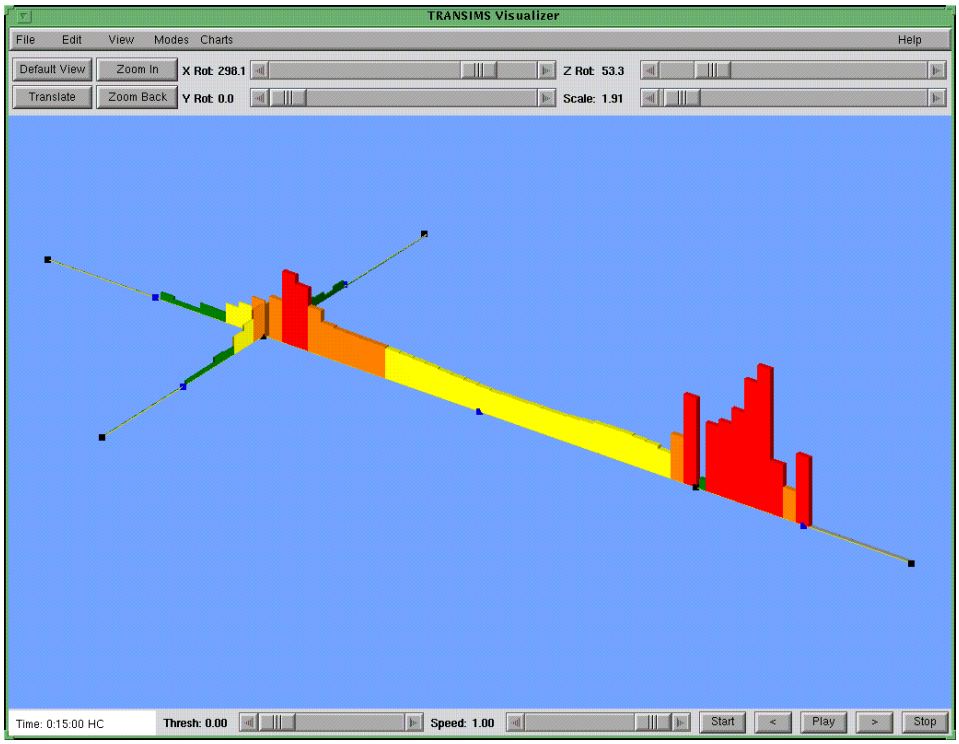

Figure 17: HC (hydrocarbon) emissions.

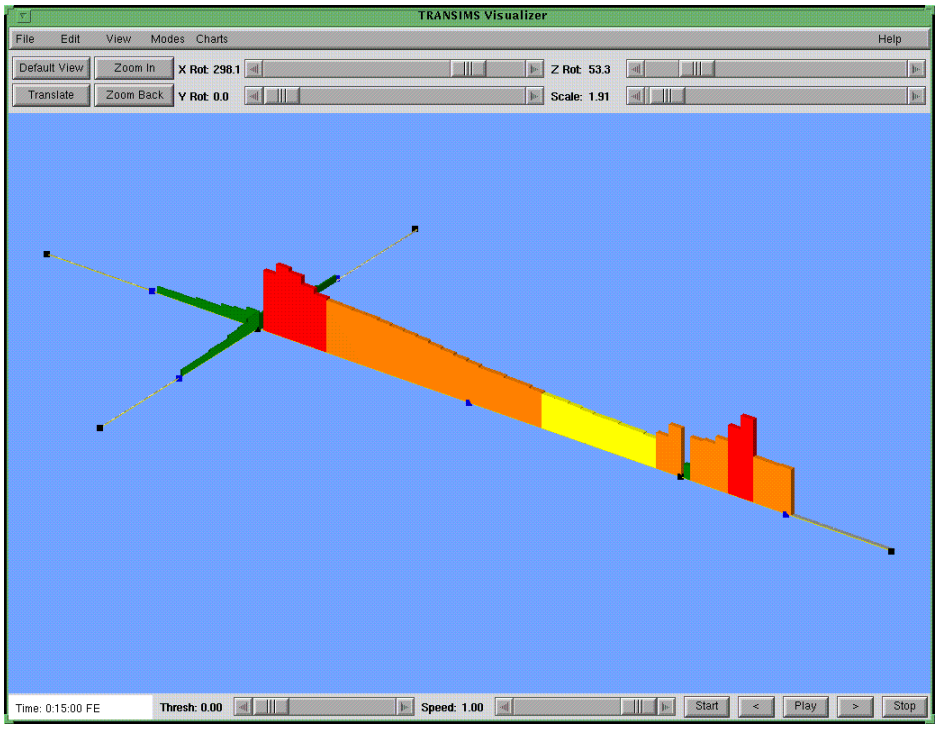

Figure 18: FE (fuel consumption).

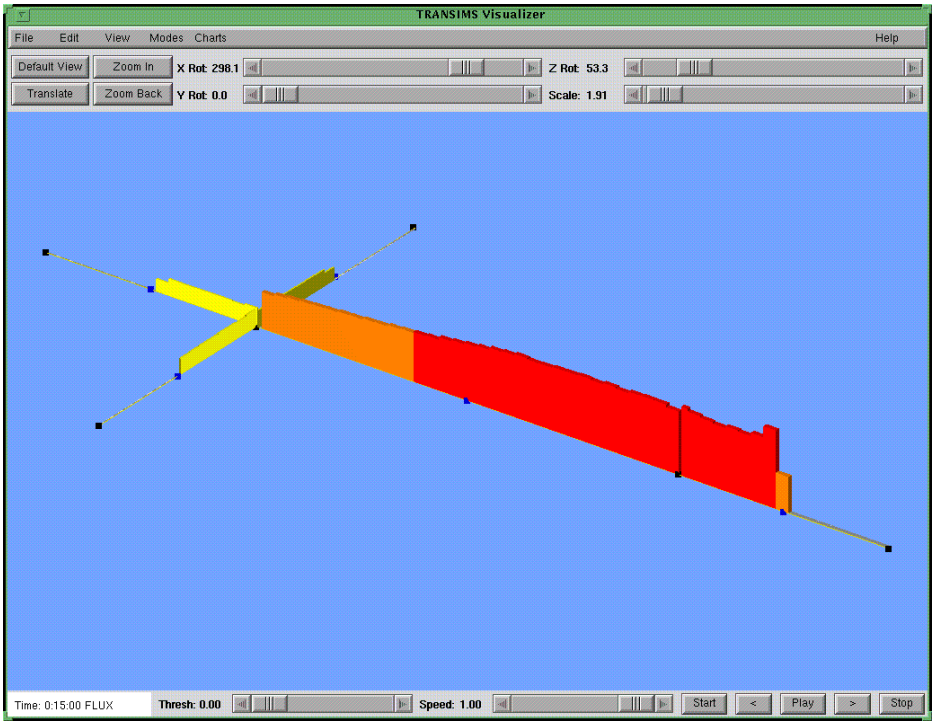

Figure 19: FLUX (vehicle flux).

10. ITERATION DATABASE

This section describes the iteration database, which records summary data for each execution of a TRANSIMS program.

10.1 Terms

mmapped Memory mapped; files are mapped directly into memory.

iteration Execution of one TRANSIMS program leg (e.g., Activity Generator, Route Planner, Traffic Microsimulator).

10.2 File Format

Two types of files will be used by the ITDB: the index file and the iteration file. The index file is described in Section 11. The primary key is given as records are added to the database (e.g., traveler ID), and the secondary key is the iteration number.

The iteration file is a text file with comma-separated fields. The meaning of the fields is determined when the file is created. The first line of each iteration file describes the iteration from which the file was generated. The second line of each file is a comma-separated list of field headings. Each field is assumed to be numeric (either integer or floating point).

10.3 Interface Functions

In any function that takes a string representing a record as an argument, an empty field is represented by two consecutive commas (i.e., “,,”). In any function that takes an array of strings representing a record as an argument, a blank field can be represented by either an empty string or a `NULL` pointer to a string. The last pointer in the array should be `NULL`.

The *String* functions return a null-terminated string that is a copy of the record/field requested. The contents of the string are modifiable, and the string needs to be freed after use.

If a particular field is empty, it is assumed that the value for that field has not changed since the last iteration.

The *Data* functions return a pointer into the mmapped file in which the record/field resides. Changing data through this pointer will change the actual iteration file where the data resides. This pointer should not be freed.

10.3.1 ITDB_Create

Signature: `ITDB* ITDB_Create(char* base_filename, char* fields)`

Description: Creates a new iteration database.

Argument: `base_filename` – filename to create files with, *filename.idx* for the index and *filename.#.it* for each iteration, where # is the iteration number.

`fields` – field names as a comma-separated string.

Return Value: A pointer to a new iteration database on iteration 0.

10.3.2 ITDB_CreateV

Signature: `ITDB * ITDB_CreateV(char* base_filename, char* fields[], int key)`

Description: Creates a new iteration database with the given filename.

`Argument:` `base_filename` – filename to create files with, *filename.idx* for the index and *filename.#.it* for each iteration, where # is the iteration number.
`fields` – field names as an array of strings.
`key` – value of the key for which to return records.

Return Value: A pointer to a new iteration database on iteration 0.

10.3.3 ITDB_Open

Signature: `ITDB* ITDB_Open(char* base_filename)`

Description: Opens an existing ITDB.

Argument: `base_filename` – the filename of the ITDB.

Return Value: A pointer to an existing iteration database on the same iteration it had when closed.

10.3.4 ITDB_Close

Signature: void `ITDB_Close(ITDB* db)`

Description: Closes an ITDB and free all resources. Upon return, `db` is no longer a valid pointer.

Argument: `db` – the database to close.

Return Value: None.

10.3.5 ITDB_CurrentIteration

Signature: `int ITDB_CurrentIteration(ITDB* db)`

Description: Returns the current iteration number.

Argument: `db` – the itdb on which to operate.

Return Value: Current iteration number.

10.3.6 ITDB_NewIteration

Signature: `int` **`ITDB_NewIteration`**`(ITDB* db, char* comment)`

Description: Starts on a new iteration.

Argument: `db` – the itdb on which to operate.
`comment` – comment to be stored as the first line of the new iteration file.

Return Value: New iteration number.

10.3.7 ITDB_Add

Signature: void **`ITDB_Add`**`(ITDB* db, int key, char* data)`

Description: Adds data to `key` for the current iteration. If data exists for the key given, the new data is added to the index following the old data.

Argument: `db` – the itdb on which to operate.
`key` – value of primary key.
`data` – a comma-separated string of field values.

Return Value: None.

10.3.8 ITDB_AddV

Signature: void **`ITDB_AddV`**`(ITDB*, int key, char* data[])`

Description: Adds data to `key` for the current iteration. If data already exists for the key given, the new data is added to the index following the old data.

Argument: `db` – the itdb on which to operate.
`key` – value of primary key.
`data` – an array of field values.

Return Value: None.

10.3.9 ITDB_GetCurrentString

Signature: `char*` **`ITDB_GetCurrentString`**`(ITDB* db, int key)`

Description: Get data for `key` from the current iteration.

Argument: `db` – the itdb in which to operate.
`key` – value of key for which to retrieve information.

Return Value: Null-terminated copy of the data. The caller is responsible for deleting this string.

10.3.10 GetCurrentData

Signature: `char* ITDB_GetCurrentData(ITDB* db, int key)`

Description: Get data for `key` from the current iteration.

Argument: `db` – the itdb on which to operate.
`key` – value of key for which to retrieve data.

Return Value: A pointer into the mmapped field. Changes to the string will change the actual file. This pointer should not be freed.

10.3.11 ITDB_GetString

Signature: `char* ITDB_GetString(ITDB* db, int it, int key)`

Description: Get data for `key` from the given iteration.

Argument: `db` – the itdb on which to operate.
`it` – iteration from which to retrieve data.
`key` – value of key for which to retrieve information.

Return Value: Null-terminated copy of the data. The caller is responsible for deleting this string.

10.3.12 ITDB_GetData

Signature: `char* ITDB_GetData(ITDB* db, int it, int key)`

Description: Get data for `key` from the given iteration.

Argument: `db` – the itdb on which to operate.
`it` – iteration from which to retrieve data.
`key` – value of key for which to retrieve information.

Return Value: A pointer into the mmapped file. Changes to the string will change the actual file. This pointer should not be freed.

10.3.13 ITDB_GetTotalString

Signature: `char* ITDB_GetTotalString(ITDB* db, int key)`

Description: Returns the latest data over all iterations for `key`. Searches back through the iterations for the last non-blank entry for each field.

Argument: `db` – the itdb on which to operate.
`key` – value of key for which to retrieve information.

Return Value: Null-terminated copy of the data. The caller is responsible for deleting this string.

10.3.14 ITDB_GetCurrentField

Signature: `char* ITDB_GetCurrentField(ITDB* db, int key, int field)`

Description: Returns the specific field for the current iteration for `key`.

`Argument:` `db` – the itdb on which to operate.
`key` – value of key for which to retrieve information.
`field` – field to retrieve.

Return Value: String containing specified field.

10.3.15 ITDB_GetField

Signature: `char* ITDB_GetField(ITDB* db, int key, int field, int it)`

Description: Returns the specified field for the specified iteration for `key`.

Argument: `db` – the itdb on which to operate.
`key` – key for which to retrieve information.
`field` – field to retrieve.
`it` – iteration from which to retrieve information

Return Value: String containing specified field.

10.3.16 ITDB_GetFirstField

Signature: `char* ITDB_GetFirstField(ITDB* db, int key, int field, int it)`

Description: Returns the specified field for the earliest iteration that has data.

Argument: `db` – the itdb on which to operate.
`key` – key for which to retrieve information.
`field` – field to retrieve.
`it` – iteration from which to retrieve information.

Return Value: String containing specified field.

10.3.17 ITDB_GetLastField

Signature: `char* ITDB_GetLastField(ITDB* db, int key, int field, int it)`

Description: Returns the specified field for the latest iteration that has data.

Argument: `db` – the itdb on which to operate.
`key` – key for which to retrieve information.
`field` – field to retrieve.
`it` – iteration from which to retrieve information.

Return Value: String containing specified field.

10.3.18 ITDB_FieldNameToNumber

Signature: `int` **`ITDB_FieldNameToNumber`**`(ITDB* db, char* name)`

Description: Converts between field name and field number.

Argument: `db` – the itdb on which to operate.
`name` – name to look up.

Return Value: Number of the given field, or –1 if it was not found.

10.3.19 ITDB_FieldNumberToName

Signature: `char*` **`ITDB_FieldNumberToName`**`(ITDB* db, int num)`

Description: Converts between field number and field name.

Argument: `db` – the itdb on which to operate.
`num` – number to look up.

Return Value: String containing the field name, or `NULL` if it was not found.

10.3.20 ITDB_ItCreate

Signature: `ITDB_It*` **`ITDB_ItCreate`**`(ITDB* db, int iteration)`

Description: Creates an iterator for the records of the given iteration.

Argument: `db` – database over which to iterate.
`iteration` – the number of the iteration over which to iterate. If `iteration` is –1, then do all iterations.

Return Value: An iterator set to the first record of the proper iteration.

10.3.21 ITDB_ItCreateRecord

Signature: `ITDB_It*` **`ITDB_ItCreateRecord`**`(ITDB* db, int key)`

Description: Creates an iterator for all iterations of the given record.

Argument: `db` – database over which to iterate.
`key` – value of the key for which to return records.

Return Value: An iterator set to the first record of the proper iteration.

10.3.22 ITDB_ItDestroy

Signature: `void ITDB_ItDestroy(ITDB_It* it)`

Description: Destroys an iterator and frees all resources.

Argument: `it` – the iterator to destroy

Return Value: None.

10.3.23 ITDB_ItReset

Signature: `void ITDB_ItReset(ITDB_It* it)`

Description: Resets iterator to beginning.

Argument: `it` – the iteration on which to operate.

Return Value: None.

10.3.24 ITDB_ItAdvance

Signature: `void ITDB_ItAdvance(ITDB_It* it)`

Description: Advances to the next record.

Argument: `it` – iteration which to operate.

Return Value: None.

10.3.25 ITDB_ItMoreData

Signature: `int ITDB_ItMoreData(ITDB_It* it)`

Description: Is there more data?

Argument: `it` – the iteration on which to operate.

Return Value: 0 if there is no more data.
non-zero if there is more data.

10.3.26 ITDB_ItGetString

Signature: `char* ITDB_ItGetString(ITDB_It* it)`

Description: Returns the current record.

Argument: `it` – the iteration on which to operate.

Return Value: A null-terminated string containing a copy of the record. The caller is responsible for freeing this data.

10.3.27 ITDB_ItGetData

Signature: `char* ITDB_ItGetData(ITDB_It* it)`

Description: Returns the current record.

Argument: `it` – the iteration on which to operate.

Return Value: A pointer into the mmapped file. Changes to the string will change the actual file. This pointer should not be freed.

10.3.28 ITDB_StringToArray

Signature: `char** ITDB_StringToArray(char* str)`

Description: Converts a single string containing multiple fields to an array of strings containing single records.

Argument: `str` – a string containing comma-separated fields.

Return Value: An array of strings, one field per string. The last element of the array is `NULL`. The caller is responsible for freeing the returned pointer.

10.3.29 ITDB_ArrayToString

Signature: `char* ITDB_ArrayToString(char** array)`

Description: Convert an array of fields to a single string.

Argument: `array` – an array of strings containing fields. The last element of the array must be set to `NULL`.

Return Value: A single string containing the comma-separated fields.

10.4 Data Structures

10.4.1 ITDB

This structure contains all of the information about an iteration database.

```
typedef struct itdb_s
{
/** The current iteration number. **/
int iteration;

/** Used to construct the itdb filename. **/
char* base_filename;

/** Name of the current iteration file; base.#.it. **/
char* idx_filename;

/** File descriptor for current iteration file. **/
```

```
int it_fd;

/** Array of labels for the fields of the database. **/
char* field_labels;

/** The number of fields. **/
int num_fields;

/** End of the current iteration file. **/
size_t it_pos;

/** Index of all iteration files. **/
BTree* index;
} ITDB;
```

10.4.2 ITDB_It

This structure is an iterator into an iteration database.

```
typedef struct itdbit_s
{
/** The index for this iterator. **/
BTree* index;

/** The index iterator. **/
BTreeIt* index_it;

/** The iteration to iterate through.  –1 means all iterations. **/
int iteration;

/** Iterate through one record only.  –1 means all records. **/
int key;

} ITDB_It;
```

10.5 Utility Programs

10.5.1 ITDB_TEST

This utility tests ITDB functions.

10.6 Files

Table 96: Iteration database library files.

Type	File Name	Description
Binary Files	libitdb.a	TRANSIMS interfaces library
Source Files	itdb.h	Defines iteration database data structures and interface functions
	itdb.c	Iteration database interface functions source file

11. INDEXING

TRANSIMS data files (particularly the activity, plan, output, and iteration database files) may be very large. Furthermore, the following common operations on these files must be efficient:

- modify small, randomly scattered records
- merge modifications back into the original file
- sort on several different keys
- retrieve specified records

File indexing provides a mechanism for efficient use of these large files.

TRANSIMS provides a C library that supports accessing files through an associated index. It also incorporates a particular strategy for using this library within the TRANSIMS framework. This section describes the indexes, library routines, and the way they are used within TRANSIMS.

11.1 Terms

index entry An index entry (the structure `BTreeEntry` defined in *btree.h*) contains a pointer to a disk file, a byte offset into the file, and the value of a major and minor key associated with the data record to be found at the given offset in the given file.

index An index (the structure `BTree` defined in *btree.h*) is a sorted set of entries together with a list of file names referred to by the entries. It is stored on disk and read into memory for use.

index file File containing a sorted index of one or more data files.

iterator An iterator (the structure `BTreeIt` defined in *btree_it.h*) is, in effect, a pointer to an index entry. It is used to iterate through an index in a fixed order.

notional file The file that would result if the data records referred to by all of the entries in an index were gathered into a single file.

11.2 Usage

An index must be created for each file to be accessed by index. Creating an index involves reading each data record in the file, determining the values of the fields to be used as keys, noting the byte offset for the beginning of that record, and inserting an entry into the index (`BTree`). Each index is given a name derived by adding an extension to the base data file. The extension indicates the major sort key for the index and that the file is an index. For example, *.trv.idx* indicates that the file is an index whose major sort key is traveler ID. These extensions are defined in the IO library header files.

Indexes are sorted according to the fields used for the major and minor sort keys. If a data file must be accessed in a particular order, for example by traveler ID, it is more efficient to build an index with that field as the major sort key than to create another data file that has been sorted. Thus, the framework will often expect several different indexes for each data file.

TRANSIMS provides C library routines for creating the indexes used by the framework, as well as standalone utility programs. Given the name of a data file to index, these routines first determine whether the required index files already exist, with a modification date more recent than that of the data file. If so, nothing is done. Where possible, these routines also create an index by examining other available indexes instead of scanning the entire data file. For example, there are two indexes for plan files; one has traveler ID as a major sort key and departure time as the minor key; the other has the sort keys reversed. Thus, one index can be created from the other without looking at the original data.

The user has access to functions used to compare keys. The current functions compare the primary sort key first. If these are equal, they compare the secondary sort keys. It is possible to specify a *don't care* value for the secondary sort key, which will compare equal to any secondary sort key value.

Indexes may be merged. In this case, entries appearing later in the set of indexes replace earlier entries. None of the data in the original data files needs to be moved to merge the indexes, yet iterating through the merged index will yield the same results as if the data files themselves had been merged and sorted.

Similarly, removing entries from an index makes the corresponding data invisible to users accessing the data file through the index.

After several merge, sort, and filter operations, it becomes difficult to determine the contents of the resulting "notional" file except by using the indexing scheme. To support users who may wish to use other data processing tools, TRANSIMS provides the ability to *defragment* the data pointed to by an index. That is, it provides executables that will create a new file on disk identical to the notional file.

Table 97: Indexes used by TRANSIMS components.

Data File Type	Extension	Major, Minor Sort Keys	Creator(s)	User(s)
Activity file	.hh.idx	Household ID, Person ID	IndexActivityFile	Route Planner, Iteration Database
Plan file	.trv.idx	Traveler ID, Activation Time	Route Planner, IndexPlanFile	Traffic Microsimulator, Iteration Database
Plan file	.tim.idx	Activation Time, Traveler ID	PlanFilter, IndexPlanFile	Traffic Microsimulator
Event Output	.trv.idx	Traveler ID, Trip ID	Iteration Database	Iteration Database
Event Output	.loc.idx	Location ID, Traveler ID	Iteration Database	Iteration Database
Vehicle file	.veh.idx	Vehicle ID, Household ID	Population Synthesizer, IndexVehicle File	Route Planner, Traffic Microsimulator

11.3 Interface Functions

11.3.1 BTree_Create

Signature: `void BTree_Create(BTree* tree, const char* data_file, const char* index_file)`

Description: Creates a new index; does not add any entries to the index file.

Argument: `tree` – tree to create; assumes `tree` is a valid pointer.
`date_file` – name of file where the data resides.
`index_file` – name of index file to create.

Return Value: None.

11.3.2 BTree_Open

Signature: `void BTree_Open(BTree* tree, const char* index_file)`

Description: Opens an existing btree index file.

Argument: `tree` – tree to open; assumes `tree` is a valid pointer.
`index_file` – name of index file to open.

Return Value: None.

11.3.3 BTree_Close

Signature: `void BTree_Close(BTree* tree)`

Description: Closes a btree and releases resources.

Argument: `tree` – tree to close; the pointer is not freed.

Return Value: None.

11.3.4 BTree_CreateFrom File

Signature: `BTree* BTree_CreateFromFile(const char* data_file, const char* index_file, enum act_keys key1, enum act_keys key2)`

Description: Creates a btree from a given data file.

Argument: `data_file` – datafile from which to read entries.
`index_file` – index file to which entries will be added.
`key1` – field number of primary key.
`key2` – field number of secondary key.

Return Value: A new index containing the entries from the data file.

11.3.5 BTree_AddFileToIndex

Signature: `void BTree_AddFileToIndex(BTree* tree, char* data_file)`

Description: Adds entries in file to tree.

Argument: `tree` – tree to which entries will be added.
`data_file` – data file from which to take entries.

Return Value: None.

11.3.6 BTree_Insert

Signature: `void BTree_Insert(BTree* tree, BTreeEntry* entry)`

Description: Inserts an entry into a btree.

Argument: `tree` – index to which entries will be added.
`entry` – the entry to add.

Return Value: None.

11.3.7 BTree_AddFilename

Signature: `int BTree_AddFilename(BTree* tree, char* filename)`

Description: Adds an additional data filename.

Argument: `tree` – tree to which filename will be added.
`filename` – date file to add.

Return Value: The file number of the added filename.

11.3.8 BTree_GetFilename

Signature: `char* BTree_GetFilename(BTree* tree, int i)`

Description: Converts from file number in a `BTreeEntry` to file name.

Argument: `tree` – tree in which to do the lookup.
`i`– file number to look up.

Return Value: The filename of the corresponding data file, or `NULL` if there is no such data file.

11.3.9 BTree_GetFileNumber

Signature: `int BTree_GetFileNumber(BTree* tree, const char* filename)`

Description: Converts from file name to file number.

Argument: `tree` – tree in which to do the lookup.
`filename` – data file name to look up.

Return Value: The filenumber of the corresponding data file, or –1 if there is no such data file.

11.3.10 BTree_ClearFilename

Signature: `void` **`BTree_ClearFilename`**`(BTree* tree)`

Description: Removes all filenames.

Argument: `tree` – tree from which to remove filenames.

Return Value: None.

11.3.11 BTree_RenumberFiles

Signature: `void` **`BTree_RenumberedFiles`**`(BTree* tree, int dest, int src)`

Description: Renumbers filenumber in entries of a tree.

Argument: `tree` – tree in which to do the renumbering.
`dest` – the new file number.
`src` – the old file number, if –1 renumber all entries.

Return Value: None.

11.3.12 BTree_GetDataPointer

Signature: `char*` **`BTree_GetDataPointer`**`(BTree* tree, BTreeEntry* e)`

Description: Gets entry in the data file for entry.

Argument: `tree` – tree in which to do lookup.
`e` – entry for which to find data.

Return Value: A pointer into the mmaped file, or `NULL` if the data is not found.
The pointer is not null-terminating ('\0'). Any changes made through this pointer will be reflected in the data file. This pointer should not be freed.

11.3.13 BTree_GetDataLine

Signature: `char*` **`BTree_GetDataLine`**`(BTree* tree, BTreeEntry* e)`

Description: Gets entry in the data file for entry.

Argument: `tree` – tree in which to do lookup.
`e` – entry for which to find data.

Return Value: A copy of the data, or `NULL` if the data is not found.
The pointer is null-terminated ('\0'). Any changes made through this pointer will not be reflected in the data file. The caller is responsible for freeing this pointer.

11.3.14 BTree_FindEntry

Signature: `BTreeEntry* BTree_FindEntry(BTree* tree, BTreeEntry* e)`

Description: Finds an entry in a tree.

Argument: `tree` – the tree in which to do the search.
`e` – entry to find, only needs keys to be set up correctly.

Return Value: The complete entry in the tree, or `NULL` if the entry was not found.

11.3.15 BTree_Validate

Signature: `void BTree_Validate(BTree* tree, const char* from)`

Description: Validates a tree. Currently, checks for the following:
- Proper order of elements in tree
- Correct number of entries
- Stuff in valid subtree
  - valid key types
  - valid file number
  - valid child pointers

Argument: `tree` – tree to validate.
`from` – where called from, used to print message (only if problem found).

Return Value: None.

11.3.16 BTreeDeleteEntry

Signature: `void BTree_DeleteEntry(BTree* tree, BTreeEntry* e)`

Description: Deletes an index entry in a tree. Does not modify any data files.

Argument: `tree` – tree from which to delete.
`e` – entry to delete.

Return Value: None.

11.3.17 BTreeIt_Create

Signature: `BTreeIt* BTreeIt_Create(BTree* tree)`

Description: Creates an iterator to a tree.

Argument: `tree` – the tree into which to point.

Return Value: An iterator into the tree. This iterator should be destroyed with **`BTreeIt_Destroy()`** to free all resources. This iterator is invalid if the tree is modified.

11.3.18 BTreeIt-Reset

Signature: `void` **`BTreeIt_Reset`**`(BTreeIt* it)`

Description: Resets an iterator to point to the first entry of the tree.

Argument: `it` – the iterator to reset.

Return Value: None.

11.3.19 BTreeIt_Advance

Signature: `void` **`BTreeIt_Advance`**`(BTreeIt* it)`

Description: Advances the iterator to the next entry in the tree.

Argument: `it` – the iterator to advance.

Return Value: None.

11.3.20 BTreeIt_MoreData

Signature: `int` **`BTreeIt_MoreData`**`(BTreeIt* it)`

Description: Are we at the end of the index?

Argument: `it` – the iterator to check.

Return Value: 0 if there are no more entries; non-zero if there are more entries.

11.3.21 BTreeIt_Get

Signature: `BTreeEntry*` **`BTreeIt_Get`**`(BTreeIt* it)`

Description: Gets the entry to which the iterator points.

Argument: `it` – the iterator to query.

Return Value: A pointer to the current entry in the tree, or `NULL` if the iterator is invalid. The entry should not be modified or freed.

11.3.22 BTreeIt_Destroy

Signature: `void BTreeIt_Destroy(BTreeIt* it)`

Description: Destroys an iterator and frees all resources.

Argument: `it` – the iterator to destroy.

Return Value: None.

11.3.23 BTreeIt_GetIterator

Signature: `BTreeIt* BTreeIt_GetIterator(BTree* tree, BTreeEntry* e)`

Description: Returns an iterator pointing to an entry in the tree.

Argument: `tree` – tree in which to find the iterator.
`e` – entry to set the iterator to, only needs keys to be set up correctly.

Return Value: An iterator that points to `e`; or `NULL` if `e` was not found.

11.3.24 BTreeIt_Compare_Equal

Signature: `int BTreeIt_Compare_Equal(BTreeIt* i1, BTreeIt* i2)`

Description: Compares two iterators.

Argument: `i1, i2` – iterators to compare.

Return Value: 0 if the iterators do not point to the same entry in the tree; non-zero if they do pint to the same entry.

11.4 Data Structures

11.4.1 Key

This structure is used to represent the value of a key.

```
typedef union u_key
{
/** A key can be either an integer or a floating point number. **/
int I;
float f;

} Key;
```

11.4.2 BTreeEntry

This structure is used as an index entry; it holds two keys—the file number and offset where the data resides.

```
typedef struct btree_entry_s
{
/** Primary Key. **/
Key key1;
```

```
/** Secondary Key. **/
Key key2;

/** Number of bytes from beginning of file. **/
off_t offset;

/** Number of data file. **/
short file;

/** Key data types. **/
char key_type;

/** Unused. **/
char pad;

} BTreeEntry
```

11.4.3 BTreeNode

This structure is used as the node of a btree; it holds up to `BTREE_ORDER` entries and `BTREE_order+1` children.

```
typedef struct btree_node_s
{
/** Number of keys currently in this node. **/
int keys;

/** Is this a leaf node? **/
int leaf;

/** Data to be stored. **/
struct btree_entry key [BTREE_ORDER];

/** Child pointers. **/
off_t child[BTREE_ORDER+1];

/** Padding to make node even multiple of page size. **/
char pad[20];

} BTreeNode
```

11.4.4 BTree

This structure contains information about a btree. It is sized so that it takes up the first page of the btree index file (`BTREE_PAGESIZE` bytes). One btree can have up to 255 data files, with a combined filename length of 5596 bytes.

```
typedef struct btree_s
{
/** Index of Root of tree. **/
off_t root;

/** Index file. **/
int index_fd;

/** Start of node array. **/
```

```
struct btree_node* index;

/** Number of nodes used. **/
size_t size;
/** Number of nodes allocated. **/
size_t allocated;

/** Number of entries in the tree. **/
size_t entries;

/** Height of the tree. **/
size_t height;

/** Field number of key1. **/
short key1;

/** Field number of key2. **/
short key2;

/** Order of this btree, used as sanity check. **/
short order;

/** Number of data files. **/
char num_filenames;

/** Version of btree file, used as sanity check. **/
char version;

/** File Descriptors for data files. **/
int data fd[256];

/** Pointers to mmaped files. **/
char* data[256];

/** Offset in filename array of filenames. **/
short filename_off[256];

/** Names of index files. **/
char filename[5596];

} BTree;
```

11.4.5 BtreeIt

This structure holds a pointer into a btree index.

```
typedef struct btree_it
{
/** Tree into which this iterator points. **/
BTree* tree;

/** Height of the tree. **/
int height;

/** Level in the tree of the iterator. **/
int level;

/** Path from root of tree to current position. **/
off_t* node;
```

```
/** Current key number at each level in path. **/
size_t* key;

} BTreeIt;
```

11.5 Utility Programs

11.5.1 IndexFilenames

The purpose of this tool is to allow easy inspection and reassignment of the data file names referred to by an index.

Each index file maintains a directory listing the names of the data files to which its entries refer, and a default UNIX directory path which is prepended to any filenames which do not begin with the character "/". The directory entries themselves contain pointers into this list of filenames. When a data file is moved, it is more efficient to update the list of filenames than to recreate the index.

This tool can be invoked in either "write" or "read" mode. In write mode, it simply prints the default directory and file names, one per line, into a file. In read mode, it reads the default directory and file names from a file and overwrites the current settings in the index file.

Usage: `IndexFilenames <index> <command> <file>`

Where `<index>` is the index file to read or modify, `<command>` is "w" to write the names of the data files to `<file>` or "r" to read the names of the data files from `<file>`. The first line of `<file>` is the default directory, which will be prepended to any data file name that does not begin with a "/" or ".". For example, to change the name of location of the datafiles for the local activities household index, the following commands would be needed:

```
IndexFilenames local.act.hh.idx w names
vi names # edit names of data files
IndexFilename  local.act.hh.idx r names
```

Example:

This example shows how to update the index *plans.tim.idx* if the data files it refers to are moved from */tmp* to */home/eubank*.

```
gershwin 1> $TRANSIMS_HOME/bin/IndexFilenames plans.tim.idx w names
gershwin 2> cat names
/tmp
plans.1
plans.2
gershwin 3> cat > newnames
/home/eubank
plans.1
plans.2
gershwin 4> $TRANSIMS_HOME/bin/IndexFilenames plans.tim.idx r newnames
```

Troubleshooting:

It is an error to reduce the number of filenames held in an index's directory, since some entries will no longer point to a valid file name. It is not an error to have duplicate file names, although it may cause inefficient memory use when the index is used.

11.5.2 IndexActivityFile, IndexPlanFile, IndexVehFile

Create appropriate indices for activity files, route plans, and TRANSIMS vehicle files. These programs are described in Section 4.5.

11.5.3 MergeIndices

The purpose of the *MergeIndices* tool is to merge and update potentially large data files without touching all the data on disk. For example, a 100 Megabyte plan file can be merged with another 100 Megabyte plan file and the result sorted by both departure time and traveler ID simply by merging and sorting the indexes for each file properly.

For each input index specified on the command line, copy the desired entries from that index into an output index. Only those entries whose primary key has not been seen in a previously processed index are desired. The input indexes are processed from last to first, so this restriction essentially means that entries from indexes specified later on the command line overwrite those specified earlier on the command line.

Usage:

```
MergeIndices <output-name> <index1> [<index2> [<index3> ... ]]
```

Example:

The following command will merge the indexes for transit driver plans stored in the file *plans.transit*, plans from the first iteration of the Router stored in *plans.pop.1*, and plans from the second iteration of the Router stored in *plans.pop.2*:

```
MergeIndices out.trv.idx plans.transit.trv.idx plans.pop.1.trv.idx plans.pop.2.trv.idx
```

The output index will be *out.trv.idx*. Assuming all the transit driver IDs are distinct from other members of the population, *out.trv.idx* will contain all of the transit driver plans, all of the plans from *plans.pop.2*, and plans for all of the travelers in *plans.pop.1* who did not appear in *plans.pop.2*.

The resulting index can be used to create an index sorted by time using the *IndexPlanFile* tool. These indexes can be used directly by the Traffic Microsimulator (or distributed using the *DistributePlans* tool, or viewed using the *PlanFilter* tool) without the need to create an actual file *out* containing all the data for the plan legs. If desired, such a file could be created using the *-d* option of the *PlanFilter* tool.

Troubleshooting:

Only the primary key is used to distinguish entries. Thus, *MergeIndices* works well for plans indexed by traveler ID, but not for plans indexed by departure time. Similarly, if the household ID is used as a key, all travelers in a household should be updated at once.

11.5.4 IndexDefrag

Defragment and merge the datafiles for an index. The entries in an index are written to a new datafile in the order that they appear in the index. The index is modified to use the new data file. For example, if *vehicles.hh.idx* refers to *vehicles1*, and *vehicles2*, then the command

```
IndexDefrag vehicles.hh.idx vehicles.new
```

will create a new datafile, with the entries from *vehicles1* and *vehicles2* that occur in *vehicles.hh.idx*. The index file *vehicles.hh.idx* will now refer only to file *vehicles.new*.

11.6 Files

Table 98: Indexing library files.

Type	File Name	Description
Binary Files	libTIO.a	TRANSIMS Interfaces library
Source Files	btree.h	Defines `Btree` and `BTreeEntry` data structures and interface functions
	btree.c	*Btree.h* interface functions source file
	btree_it.h	Defines BtreeIt data structure and interface functions
	btree_it.c	*btree_it.h* interface functions source file

11.7 Examples

```
#include "IO/btree.h"
#include "IO/btree_it.h"

int main(int argc, char* argv[])
{
  char* data_file;
  char* index_file;
  BTreeEntry entry;
  BTree* tree;
  BTreeIt *it;

  index_file = "sample0.idx";
  data_file = "sample1.dat";

  /* Create an index file */
  tree = BTree_CreateFromFile(data_file,
                              index_file,
                              kActivityPerson,
                              kActivityStartMin);

  /* Add a second data file to the index */
  data_file = "sample2.dat";
  BTree_AddFileToIndex(tree, data_file);

  /* Delete an entry */
  entry.key1.i = 0;
  entry.key2.f = 0.0;
  entry.key_type = K_IF;
  BTree_DeleteEntry(tree, &entry);

  /* Use an iterator to examine each entry */
```

```
  it = BTreeIt_Create(tree);
  BTreeIt_Reset(it);
  while (BTreeIt_MoreData(it))
  {
    BTreeEntry* e;
    BTreeEntry* e2;
    BTreeIt* it2;
    /* Get the entry for this iterator */
    e = BTreeIt_Get(it);

    /* Get a second iterator, pointing to the same entry */
    it2 = BTreeIt_GetIterator(tree, e);
    /* Get the entry for this iterator */
    e2 = BTreeIt_Get(it2);
    /* Verify that the entries are the same (they should be) */
    if (!BTree_Compare_Equal(e, e2) || !BTreeIt_Compare_Equal(it, it2))
    {
      if (!BTree_Compare_Equal(e, e2))
        printf("Entries differ\n");
      if (!BTreeIt_Compare_Equal(it, it2))
        printf("Iterators differ\n  ");
    }
    /* Clean up the second iterator */
    BTreeIt_Destroy(it2);

    /* Advance to the next entry */
    BTreeIt_Advance(it);
  }
  BTreeIt_Destroy(it);

  BTree_Close(tree);
  free(tree);
  return 0;
}
```

12. VISUALIZATION

This section describes the file formats used as input into the Output Visualizer.

12.1 Terms

Variable Size Box Format A box of any size and location on a given link is described by this format.

Constant Size Box Format Data for each box of a given fixed size is described by this format.

Vehicle Evolution Format Data on vehicle position, type, passengers, and velocity is described by this format.

12.2 File Format

12.2.1 Variable Size Box Format

Fields in the variable size box format are tab-delimited.

Each line of the variable size box format contains at least six mandatory fields:
1) TIME
2) Link ID
3) Node ID
4) Distance – the distance where the described box ends from the beginning of the link.
5) Length – the total length of the box being described.
6) Data value

Additionally, one may add up to nine more data value columns. It is suggested that one provide a labeling line on the first line of the file describing each column as shown below.

```
TIME LINK NODE DISTANCE LENGTH DataVal1 DataVal2 DataVal3 DataVal4...
```

Format:

```
<TIME> <Link ID> <Node ID> <Distance> <Length> <Data Value 1> [<Data Value 2> ... <Data Value 10>]
```

Example:

```
TIME LINK NODE DISTANCE LENGTH DataVal1 DataVal2 DataVal3

800     1400   1256    24.75            12.50        10.0         20.4         35.6
```

At time 800 of the simulation, a box should be drawn of length 12.5 that ends 24.75 meters from node 1256 of link 1400. The data values for each of the first three columns are 10.0, 20.4, and 35.6 respectively.

12.2.2 Constant Size Box Format

The Constant Size Box Format (Table 99) is a binary file format and consists of the following fields in the given order.

Table 99: Constant Size Box Format data structure fields.

Field	Description	Allowed Values
Time	Current simulation time for which this data has been collected.	Integer (32 bits)
Count	Number of vehicles that have passed through during the sampling time.	Integer (32 bits)
Link	Link id for the current box.	Integer (32 bits)
Sum	Sum of all velocities for vehicles passing through this box during the sampling time.	Decimal (32 bits)

The constant size box format file should be sorted by the time field.

12.2.3 Vehicle Evolution Format

The Vehicle Evolution Format (Table 100) is a binary file format consisting of a single data structure type shown below.

Table 100: Vehicle Evolution Format data structure fields.

Field	Description	Allowed Values
Status	Vehicle type number in the lower 8 bits, and the number of passengers in the upper 8 bits.	Integer (16 bits)
Theta	Number of degrees from due east the vehicle is pointed. The angle is calculated counterclockwise from due east.	Integer (16 bits)
Time	Current simulation time for which this current record has been collected.	Integer (32 bits)
Velocity	Current velocity of the vehicle.	Decimal (32 bits)
X	Current *x* position of the front middle of the vehicle.	Decimal (32 bits)
Y	Current *y* position of the front middle of the vehicle.	Decimal (32 bits)
Z	Current *z* position of the front middle of the vehicle.	Decimal (32 bits)
Vehicle ID	Vehicle ID.	Integer (32 bits)
Link ID	Current link ID on which the vehicle is traveling.	Integer (32 bit)

The vehicle evolution file should be sorted by time.

12.3 Utility Programs

12.3.1 vehtobin

The *vehtobin* program converts IOC-2 text format to the binary format required by the Output Visualizer. Usage is as follows:

```
vehtobin inputfilename outputfilename
```

12.4 Files

Table 101: Visualization library files.

Type	File Name	Description
Binary Files	libTIO.a	TRANSIMS Interfaces library
Vehtobin Source Files	vehtobin.h	Defines data structures and interface functions to convert IOC-2 text data files into binary vehicle evolution files for use with the Output Visualizer.
	vehtobin.c	Main function to convert IOC-2 text data files into binary vehicle evolution files for use with the Output Visualizer.

13. CONFIGURATION

This section describes the format of configuration files. These files contain the parameters used by the various TRANSIMS software modules.

13.1 Terms

key Character string (containing no spaces) representing a configuration parameter.

value Number or character string.

13.2 File Format

Configuration files are text files that contain lines of the following types:

- A key followed (optionally) by a value and (optionally) by a comment starting with the pound (#) symbol. The key and the value must be separated by space and/or tab characters.
- A comment line staring with the pound symbol (#).
- A blank line.

13.3 Interface Functions

Functions are available for reading and writing records of a configuration file.

13.3.1 ConfigRead

Signature: `int ConfigRead(FILE* file, TConfigRecord* record)`

Description: Read a record from a configuration file.

Argument: `file` – `FILE` pointer for the configuration file.
`record` – pointer to `TConfigRecord` structure into which the record is read.

Return Value: Nonzero if the record was successfully read, or zero if not.

13.3.2 ConfigWrite

Signature: int **ConfigWrite**(FILE* file, const TConfigRecord* record)

Description: Write a record to a configuration file.

Argument: `file` – `FILE` pointer for the configuration file.
`record` – pointer to TConfigRecord structure from which the record is written.

Return Value: Nonzero if the record was successfully written, or zero if not.

13.4 Data Structures

13.4.1 TConfigRecord Structure

Structure for configuration file records.

```
typedef struct
{
/**  The key, if the record has one. **/
INT8 fKey[64];

/**  The value, if the record has one. **/
INT8 fValue[256];

/**  The comment, if the record has one. **/
INT8 fComment[512];

} TConfigRecord;
```

13.5 Utility Programs

13.5.1 SetEnv

The *SetEnv* program takes the keys in a configuration file and converts them into UNIX shell environment variables set to the values corresponding to the keys. Its first argument is the name of the UNIX shell and its second argument is the name of the configuration file; it does not recurse nested configuration files. It is typically used as follows:

```
eval `SetEnv csh default.config`
eval `SetEnv csh my-run.config`
```

where `default.config` is the default configuration file identified in the configuration file `my-run.config`.

13.6 Files

Table 102: Configuration library files.

Type	File Name	Description
Binary Files	libTIO.a	TRANSIMS Interfaces library
Utilities	SetEnv	Environment variable setting utilities
Source Files	configio.h	Defines configuration file data structures and interface functions
	configio.c	Configuration file interface functions source file

13.7 Configuration Keys

The configuration key CONFIG_DEFAULT_FILE specifies the name of a configuration file whose keys and values are to be used in cases where a key is not set in the current configuration file.

13.8 Examples

Figure 20 and Figure 21 give examples of typical configuration and default configuration files, respectively. Note that when keys are duplicated in these files, the value in the non-default file takes precedence.

Figure 20: Example configuration file.

```
CONFIG_DEFAULT_FILE /home/transims/allstr-run/default.config

NET_PROCESS_LINK_TABLE Process_Link.minimal.tbl

ROUTER_MAX_DEGREE 15

CA_BIN /home/projects/transims/config/integration/bin/ARCH.PVM.SUN4SOL2/CA
CA_SIM_STEPS 7200
CA_MASTER_MESSAGE_LEVEL 1

PAR_COMMUNICATION   PVM
PAR_SLAVES 1
```

Figure 21: Example default configuration file.

```
##################### GLOBAL PARAMETERS #####################

# The width of a lane in meters
# float
GBL_LANE_WIDTH 3.5

# The length of a cell in meters
# float
GBL_CELL_LENGTH 7.5


##################### NETWORK PARAMETERS ####################

NET_DIRECTORY                /home/transims/allstr-run/network/

NET_NODE_TABLE               Node.tbl
NET_LINK_TABLE               Link.tbl
NET_POCKET_LANE_TABLE        Pocket_Lane.tbl
NET_LANE_USE_TABLE           Lane_Use.tbl
NET_SPEED_TABLE              Speed.tbl
NET_LANE_CONNECTIVITY_TABLE  Lane_Connectivity.tbl
NET_TURN_PROHIBITION_TABLE   Turn_Prohibition.tbl
NET_UNSIGNALIZED_NODE_TABLE  Unsignalized_Node.tbl
NET_SIGNALIZED_NODE_TABLE    Signalized_Node.tbl
NET_PHASING_PLAN_TABLE       Phasing_Plan.tbl
NET_TIMING_PLAN_TABLE        Timing_Plan.tbl
NET_SIGNAL_COORDINATOR_TABLE Signal_Coordinator.tbl
NET_DETECTOR_TABLE           Detector.tbl
```

```
NET_BARRIER_TABLE            Barrier.tbl
NET_PARKING_TABLE            Parking.tbl
NET_TRANSIT_STOP_TABLE       Transit_Stop.tbl
NET_ACTIVITY_LOCATION_TABLE  Activity_Location.tbl
NET_PROCESS_LINK_TABLE       Process_Link.tbl
NET_STUDY_AREA_LINKS_TABLE   Study_Area_Link.tbl


##################### SYNTHETIC POPULATION PARAMETERS ####################

POP_NUMBER_HH           1000
POP_BASELINE_FILE       /home/transims/allstr-run/output/allstr.basepop
POP_LOCATED_FILE        /home/transims/allstr-run/output/allstr.locpop
POP_STARTING_VEHICLE_ID 100000
POP_STARTING_HH_ID      1
POP_STARTING_PERSON_ID  101


##################### ACTIVITY GENERATOR PARAMETERS ####################

ACT_FULL_OUTPUT          /home/transims/allstr-run/output/allstr.activities
ACT_PARTIAL_OUTPUT       /home/transims/allstr-run/output/allstr.partact
ACT_FEEDBACK_FILE        /home/transims/allstr-run/output/allstr.actfeed
ACT_WORK_LOC_ALPHA       1
ACT_WORK_LOC_BETA        1
ACT_WORK_LOC_GAMMA       1
ACT_TIME_ALPHA           1
ACT_TIME_BETA            1
ACT_MODE_ALPHA           1
ACT_MODE_BETA            1
ACT_WORK_LOCATION_OPTION 1
ACT_MODE_CHOICE_OPTION   4
ACT_HOME_HEADER          HOME
ACT_WORK_HEADER          WORK
ACT_ACCESS_HEADER        ACCESS


##################### OUTPUT PARAMETERS #####################

OUT_DIRECTORY               /home/transims/allstr-run/output

OUT_SNAPSHOT_NAME_1         allstr.snapshot
OUT_SNAPSHOT_BEGIN_TIME_1   0
OUT_SNAPSHOT_END_TIME_1     86400
OUT_SNAPSHOT_TIME_STEP_1    1
OUT_SNAPSHOT_EASTING_MIN_1  1
OUT_SNAPSHOT_EASTING_MAX_1  1000000
OUT_SNAPSHOT_NORTHING_MIN_1 1
OUT_SNAPSHOT_NORTHING_MAX_1 1000000
OUT_SNAPSHOT_NODES_1        /home/transims/allstr-run/data/allstr.nodes
OUT_SNAPSHOT_LINKS_1        /home/transims/allstr-run/data/allstr.links
OUT_SNAPSHOT_SUPPRESS_1
OUT_SNAPSHOT_FILTER_1

OUT_EVENT_NAME_1            allstr.event
OUT_EVENT_BEGIN_TIME_1      0
OUT_EVENT_END_TIME_1        86400
OUT_EVENT_TIME_STEP_1       1
OUT_EVENT_EASTING_MIN_1     1
OUT_EVENT_EASTING_MAX_1     1000000
OUT_EVENT_NORTHING_MIN_1    1
OUT_EVENT_NORTHING_MAX_1    1000000
OUT_EVENT_NODES_1           /home/transims/allstr-run/data/allstr.nodes
OUT_EVENT_LINKS_1           /home/transims/allstr-run/data/allstr.links
OUT_EVENT_SUPPRESS_1
OUT_EVENT_FILTER_1

OUT_SUMMARY_NAME_1          allstr.summary
OUT_SUMMARY_BEGIN_TIME_1    0
OUT_SUMMARY_END_TIME_1      86400
OUT_SUMMARY_TIME_STEP_1     900
OUT_SUMMARY_SAMPLE_TIME_1   60
OUT_SUMMARY_BOX_LENGTH_1    150
OUT_SUMMARY_EASTING_MIN_1   1
OUT_SUMMARY_EASTING_MAX_1   1000000
```

```
OUT_SUMMARY_NORTHING_MIN_1  1
OUT_SUMMARY_NORTHING_MAX_1  1000000
OUT_SUMMARY_NODES_1         /home/transims/allstr-run/data/allstr.nodes
OUT_SUMMARY_LINKS_1         /home/transims/allstr-run/data/allstr.links
OUT_SUMMARY_SUPPRESS_1
OUT_SUMMARY_FILTER_1

##################### SIMULATION PARAMETERS #################

# see IO/log.h for possible levels
CA_SLAVE_MESSAGE_LEVEL    0
CA_MASTER_MESSAGE_LEVEL   0

# name of executable (used by Msim.pl)
CA_BIN CA

# the max number of occupants of a bus
# int > 1
CA_BUS_CAPACITY   50

# the number of cells a bus occupies in a jam
# float > 0.0
CA_BUS_LENGTH     2.0

# the acceleration of a car, bus, etc.
# (in cells per timestep per timestep)
# float > 0.0
CA_MAXIMUM_ACCELERATION     0.4
CA_BUS_MAXIMUM_ACCELERATION    0.1

# the maximum speed of a car, bus, etc.
# (in cells per timestep)
# float > 0.0
CA_MAXIMUM_SPEED     4.5
CA_BUS_MAXIMUM_SPEED    2.5

# If nonzero, no attempt will be made to read in transit vehicles
# and transit passengers will not be simulated.
# int(?)
CA_NO_TRANSIT       1

# Some time after a vehicle becomes off plan, it will exit the simulation.
# the probability that a vehicle with speed >= 1 will decelerate by 1
# (also an increment added to the speed limit on a link)
# in the discrete version (not compiled with -DCONTINUOUS)
# float > 0 and < 1
CA_DECELERATION_PROBABILITY   0.2

# use to compute the number of cells that must be vacant in an acceptable gap
# (acceptable gap is speed of oncoming vehicle * Velocity Factor)
# float (> 1.0 ? )
CA_GAP_VELOCITY_FACTOR      3.0

# Probability of proceeding when interfering gap is not acceptable
# in cases of links with competing stop/yield signs
# float > 0 and < 1
CA_IGNORE_GAP_PROBABILITY      0.66

# The number of vehicles which can be buffered in each
# of an intersection’s queues (One queue for each lane of each incoming link)
# int > 1
CA_INTERSECTION_CAPACITY     10

# Vehicles take at least this many timesteps to traverse an intersection
# int >= 0
CA_INTERSECTION_WAIT_TIME     1

# Can’t change lanes if random variable drawn on each timestep for each vehicle
# is less than this
# float > 0 and < 1
CA_LANE_CHANGE_PROBABILITY     0.99

# number of cells ahead to look for deciding which lane is best upon entering a link
# int >= 0
CA_LOOK_AHEAD_CELLS       35
```

```
# If vehicle has not moved for this many timesteps,
# it becomes off-plan and chooses a different destination link, if possible.
# int >= 0
CA_MAX_WAITING_SECONDS       600

# The exit time is the minimum of the expected arrival time at the destination
# and the current time + OFF_PLAN_EXIT_TIME
# int >= 0
CA_OFF_PLAN_EXIT_TIME   1

# Determines, in a complicated way, whether lane changes for the
# sake of following a plan need to be considered
# int >= 0
CA_PLAN_FOLLOWING_CELLS   70

# specify start time for simulation
# int
CA_SIM_START_HOUR  0
CA_SIM_START_MINUTE  0
CA_SIM_START_SECOND  0

# number of timesteps to simulate
# int >= 0
CA_SIM_STEPS  3600

# send map of locations of all accessories to all slaves
CA_BROADCAST_ACC_CPN_MAP       0

# migrate travelers by broadcasting them
CA_BROADCAST_TRAVELERS         1

# number of time-steps to be executed before slaves synchronize with master
CA_SEQUENCE_LENGTH      1

# Initialize the random seed
# seed48 is called with a pointer to the first element of an array
# of these 3 unsigned shorts
# unsigned short
CA_RANDOM_SEED1  1
CA_RANDOM_SEED2  2
CA_RANDOM_SEED3  3

# Use the cached binary representation of the network database
# in the file specified by CA_NETWORK_FILE
# int
CA_USE_NETWORK_CACHE  0
# string
# CA_NETWORK_FILE

# The following delays model just the time it takes to walk up the steps or
# through the doors or whatever. They have nothing to do with the
# time spent waiting in the queue.

# The mean number of seconds it takes a traveler to board a transit vehicle.
# float >= 0.0
CA_ENTER_TRANSIT_DELAY   1.6

# The mean number of seconds it takes to disembark.
# float >= 0.0
CA_EXIT_TRANSIT_DELAY  1.8

# The number of seconds after a vehicle reaches the stop before
# passengers can start boarding
CA_TRANSIT_INITIAL_WAIT  5

# Name of a file containing TRANSIMS format vehicle information
# (locations, type, etc.)
CA_VEHICLE_FILE  /home/transims/allstr-run/output/allstr.vehicles

CA_USE_PARTITIONED_ROUTE_FILES  0

CA_LATE_BOUNDARY_RECEPTION     1
CA_PARALLEL_LOG                0
```

```
CA_PARALLEL_IO_TEST_MODE       0
CA_PARALLEL_IO_TEST_INTERVAL   30

CA_OUTPUT_BUFFER_COUNT         32

CA_RTM_SAMPLE_INTERVAL         0


##################### TRANSIT PARAMETERS ####################

# Name of a file containing TRANSIMS format transit route information
# (list of stops for each route)
# string
TRANSIT_ROUTE_FILE /home/transims/allstr-run/data/allstr.routes

# Name of a file containing TRANSIMS format transit schedule information
# (list of arrival time for each vehicle at each stop)
# string
TRANSIT_SCHEDULE_FILE  /home/transims/allstr-run/data/allstr.schedules

##################### PLAN PARAMETERS #######################

# Name of a file containing TRANSIMS format legs
# string
PLAN_FILE  /home/transims/allstr-run/output/allstr.plans

##################### ROUTER PARAMETERS #####################

ROUTER_OUTPUT_PLAN_FILE /home/transims/allstr-run/output/allstr.plans
ROUTER_ACTIVITY_FILE    /home/transims/allstr-run/output/allstr.activities
ROUTER_VEHICLE_FILE     /home/transims/allstr-run/output/allstr.vehicles
ROUTER_MODE_MAP_FILE    /home/transims/allstr-run/data/allstr.modes

ROUTER_MAXNFASIZE         5
ROUTER_MAX_DEGREE         15
ROUTER_INTERNAL_PLAN_SIZE 400
ROUTER_VERBOSE  2

# If length < corr_thresh * dist, adjust the length
# float
ROUTER_CORR  0.0

# ??
# float
ROUTER_OVERDO  3.0

# Backdating time of travel information ??
# int
ROUTER_ZERO_BACKD  0

##################### LOGGING PARAMETERS ####################

LOG_LOG_CONFIG        0
LOG_LOAD_NETWORK      1
LOG_PARTITIONING      1
LOG_DISTRIBUTION      1
LOG_RUNTIMEMONITOR    0
LOG_CONTROL           0
LOG_TIMING            1
LOG_BOUNDARIES        0
LOG_ROUTING           1
LOG_ROUTING_DETAIL    1
LOG_TIMESTEP          1
LOG_TIMESTEP_DETAIL   1
LOG_PARALLEL          0
LOG_VEHICLES          1
LOG_MIGRATION         1
LOG_MIGRATION_DETAIL  1
LOG_TRANSIT           1
LOG_EMISSIONS         1
LOG_IO_DETAIL         0

##################### VISUALIZER PARAMETERS #################

# int, will be single buffered if non-zero
```

```
VIS_SINGLE_BUFFERED  0

# Name of a file containing batch commands (unused)
# string
VIS_BATCH_FILE

# The length of a box in meters
# float
VIS_BOX_LENGTH          150.0

##################### PARTITIONING PARAMETERS ###############

PAR_PVM_ROOT                    /sw/Cvol/pvm3
PAR_PVM_ARCH                    SUN4SOL2
PAR_PVM_WAIT_FOR_DEAMON         20

PAR_MPI_ROOT                    /sw/Cvol/mpich
PAR_MPI_ARCH                    solaris
PAR_MPI_DEVICE                  ch_p4

PAR_MIN_CELLS_TO_SPLIT         10
PAR_SLAVES                      2

# if 1, use orthogonal bisection to distribute the network
# otherwise, use the METIS graph partitioning library
# int
PAR_USE_METIS_PARTITION         1
PAR_USE_OB_PARTITION            0

PAR_PARTITION_FILE              /tmp/partition
PAR_SAVE_PARTITION              0

# if 0 use (number of lanes) for edge weight, (length * number of lanes) for edge penalty
#       and 0 for node weights in the partitioning algorithm
# otherwise, use the file named in RTM_FEEDBACK_FILE and RTM_PENALTY_FACTOR.
# int
PAR_USE_RTM_FEEDBACK            0

# Filename for edge and node weights for partitioning
# File format is lines of the form:
# 0 Id Weight
# 1 Id Weight Penalty
# The first line sets a node weight
# the second line sets an edge weight: if penalty is -1, use current value *
RTM_PENALTY_FACTOR
#                                       otherwise use Penalty * RTM_PENALTY_FACTOR
# string
PAR_RTM_FEEDBACK_FILE           /tmp/rtm

# See above for RTM_FEEDBACK_FILE
# float > 0.0
PAR_RTM_PENALTY_FACTOR         100.0

PAR_REPORT_OUTGOING_LINK_TIME_ONLY    1

##################### SELECTOR PARAMETERS ###################

# Only travelers whose (actual - expected) / expected
# is greater than this will be affected by any operations
# float > 0
SEL_FRUSTRATION_THRESH 1.5

# Fraction of travelers to select for
# just rerouting
# reassigning activities
# choosing a new mode preference
# changing the time of activities
# float, >= 0 and <= 1
SEL_REROUTE_FRAC 0.1
SEL_REASSIGN_FRAC 0.1
SEL_REMODE_FRAC 0.1
SEL_RETIME_FRAC 0.1

# Name of files in which to place traveler ids
# selected for each of the possible changes
```

```
# string
SEL_REROUTE_FILE
SEL_REMODE_FILE
SEL_RETIME_FILE
SEL_REASSIGN_FILE

# ==============================================================================
# Local Variables:
# tab-width:4
# End:
# ==============================================================================
```

14. LOGGING

The TRANSIMS logging interface is to be used for the logging output of all applications that will be part of the TRANSIMS suite of software modules and will be integrated into the development environment. Using a single interface allows the standardization of logging messages.

14.1 Terms

MSG_PRINT	Normal informative message.
MSG_WARNING	Warning that may need user attention but is most likely not to corrupt the application results.
MSG_SEVERE_WARNING	Warning that does not require the user to shut down the application but will most likely result in corrupted output.
MSG_ERROR	Actual error message that results in immediate termination of the program.

14.2 Interface Functions

Each logging message is associated with a module passed in the parameter `theSubSystem`. There are predefined modules for most subsystems in TRANSIMS (see *IO/log.h for a list*.)

There are four different message levels that are passed in the parameter `theMessageLevel`:

1) MSG_PRINT: This is a normal informative message. It does NOT describe a warning or an error.
2) MSG_WARNING: This is a warning that may need user attention, but is most likely not to corrupt the application results.
3) MSG_SEVERE_WARNING: This is a warning that does not require the user to shut down the application but will most likely result in corrupted output.
4) MSG_ERROR: This is an actual error message that results in immediate termination of the program.

The parameter `Format` contains the actual message. It is interpreted as a C-style `printf(1)` format string that permits the passing of additional parameters after the format string. There is no need to terminate the format string with a newline character, since that will be automatically added.

Notes:
1) Do not try to by-pass the interface since this may result in messages getting lost.
2) Refrain from using the strings “ERROR” or “WARNING” (or any other pattern listed in the DEFINES-Reserved String Pattern section of the *log.h* file) in your messages. The interface will add appropriate strings to your messages so that they can be identified.
3) Choose the message level with care since “harmless” levels such as MSG_PRINT or MSG_WARNING may be deactivated when the application is run in production mode. Really important messages should be of type MSG_SEVERE_WARNING or MSG_ERROR.

4) Do not make any assumption about where the logging output will end up. The default will be standard output, but it will also be redirected to a file.

```
void
cMessage(enum TSubsystem theSubSystem, enum TMessageLevel
   theMessageLevel, const char * Format, …);
```

14.3 Files

Table 103: Logging library files.

Type	File Name	Description
Binary Files	libTIO.a	TRANSIMS Interfaces library
Source Files	log.c	Source for logging functions
	log.h	Logging interface functions source file

14.4 Examples

```
cMessage (SUB_CA, MSG_WARNING, “More vehicle (%d) than expected (%d)”,
          NrOfVehicle, NrOfExpectedVehicles;
```

15. REFERENCES

[BBM 96] R. J. Beckman, K. A. Baggerley, and M. D. McKay, *Creating Synthetic Baseline Populations*, Statistics Group, Los Alamos National Laboratory, Los Alamos, NM, 1996.

[Bl] J. Blodgett, "MABLE/GEOCORR Geographic Correspondence Engine," http://www.oseda.missouri.edu/plue/geocorr/doc/article.html.

[Ce 96] K. Cervenka, personal communication, 1996.

[Do 97] R. Donnelly, personal communication, 1997.

[GHA 88] Federal Highway Administration, *Manual on Uniform Traffic Control Devices*, (Washington, D.C.: U.S. Government Printing Office, 1988).

[ITE 85] Institute of Transportation Engineers, *Traffic Control Systems Handbook*, (Washington, D.C.: ITE Publications, 1985).

[ITE] Institute of Transportation Engineers, *Traffic Detector Handbook*, (Washington, D.C.: ITE Publications, n.d.).

[MM 84] M. D. Meyer and E. J. Miller, *Urban Transportation Planning*, (New York: McGraw-Hill, 1984).

[Or 93] F. L. Orcutt, Jr., *The Traffic Signal Book*, (Englewood Cliffs, New Jersey: Prentice Hall, 1993).

[PP 93] C. S. Papacostas and P. D. Prevedouros, *Transportation Engineering and Planning*, (Englewood Cliffs, New Jersey: Prentice Hall, 1993).