S.C.R.T.D. LIBRARY

U.S. Department
of Transportation

Systan's Macro-Analytic Regionwide Transportation Model

Program Maintenance Manual

March 1983

HE
147.6
.S97
v.3

V3
S.C.R.T.D. LIBRARY

Systan's Macro-Analytic Regionwide Transportation Model: Program Maintenance Model

Final Report
March 1983

Prepared by
Andy Canfield and Wei-Yue Lim
Systan, Inc.
343 Second Street
Los Altos, California 94022

Prepared for
Office of Technical Assistance
Urban Mass Transportation Administration
Washington, D.C. 20590

and

Office of Technology and Planning Assistance
Office of the Secretary of Transportation
Washington, D.C. 20590

DOT-I-83-59

05800

HE
147.6
.S97
v.3

PREFACE

SYSTAN's Macro-Analytic Regionwide Transportation (SMART) model is a sketch planning tool for evaluating public transportation alternatives for metropolitan areas. The model and its documentation were developed as part of the Paratransit Integration Program sponsored by the Office of Bus and Paratransit Technology and the Urban Mass Transportation Administration and by the Office of Technology and Planning Assistance of the Office of the Secretary of Transportation. The Paratransit Integration Program is concerned with the development and application of macro-analytic techniques for policy and preliminary planning at the local level.

The SMART model documentation consists of three volumes:

> Application Manual: This report describes the use of the model to formulate, evaluate, and compare public transit options for urban regions. It discusses the structure of the model and the purpose of each major component. It also includes detailed application information for four case studies. The document is designed for use by transit planners who must assess the suitability of the model and, if appropriate, use it to investigate urban transportation alternatives.
>
> User's Guide: This document focuses on the preparation and formatting of data for use in the model. Examples are presented, and error messages are explained. The document builds on material in the Applications Manual and is required to run the SMART computer program.
>
> Program Maintenance Manual: This manual describes the internal structure of the computer program, including module structure and linkage and data structures. It includes material on installation and on potential model alterations. Written for the skilled FORTRAN programmer, each installation of the SMART model computer program should have at least one copy of this manual.

The SMART computer program was written by Andrew J. Canfield. Site applications were performed by Carolyn Fratessa and Dr. Wei-Yue Lim. All work was performed under the direction of Dr. Paul S. Jones. SYSTAN gratefully acknowledges the technical and administrative guidance and support of Edward Neigut and Michael Markowski of UMTA. Many other UMTA and DOT staff members have given freely of their time and skill to offer valuable input to the work. However, SYSTAN is solely responsible for the results.

CONTENTS

page

PREFACE iii

1. INTRODUCTION 1-1

Notes on the Contents 1-1
System Summary 1-1

2. DATA STRUCTURES 2-1

CBD.PARAMETERS (CBDPRM) 2-4
DEMAND 2-5
DOOR.TO.DOOR.PARAMETERS (DORPRM) 2-6
DEMAND.PARAMETERS (DPARM) 2-7
DUMPER.PARAMETERS (DPRPRM) 2-8
FEEDER.PARAMETERS (FDRPRM) 2-9
FIXED 2-10
FRITZ 2-12
INPUT.RECORD (INREC) 2-13
INPUT.OUTPUT.CONTROLS (IOCON) 2-14
LINKER.PARAMETERS (LKRPRM) 2-15
MODE.DEFINITIONS (MODDEF) 2-16
MODES.SELECTED (MODSEL) 2-18
RESULTS (RESULT) 2-21
SHORTEST.ROUTES (ROUTES) 2-22
URBAN.DEFAULT (UDEFLT) 2-24
URBAN.STRUCTURE (USTRUC) 2-26
Standard Indexes 2-29

3. PROGRAM STRUCTURE 3-1

Urban Structure Subsystem 3-1
Demand Subsystem 3-4
FEEDER Subsystem 3-6
DUMPER Subsystem 3-8
CBD Subsystem 3-8
LINKER Subsystem 3-11
DOOR.TO.DOOR Subsystem 3-11
REGION Subsystem 3-14
Results Printing Subsystem 3-16

4. MACHINE INDEPENDENCE AND INSTALLATION 4-1

Extensions to FORTRAN-66 4-1

Specific Violations of FORTRAN-66 4-2
Program Size 4-3
Accuracy 4-3

5. PROGRAM ALTERATIONS 5-1

Defaults 5-1
Size Limits 5-1
Module Replacements 5-4
Modal Analysis Modules 5-5
Demand Generation 5-6
LINKER Express Status Determination 5-7
Adding Mode Types 5-7
Adding Modal Parameters 5-8

APPENDIX A. INDIVIDUAL MODULE DESCRIPTIONS A-1

APPENDIX B. MODULE CROSS-REFERENCE B-1

APPENDIX C. DATA STRUCTURE REFERENCES C-1

LIST OF EXHIBITS

Figure page

2.1. PRINCIPAL INFORMATION PATHS 2-3

3.1. PROGRAM STRUCTURE SYMBOLS 3-2

3.2. URBAN STRUCTURE SUBSYSTEM 3-3

3.3. DEMAND SUBSYSTEM 3-5

3.4. FEEDER SUBSYSTEM 3-7

3.5. DUMPER SUBSYSTEM 3-9

3.6. CBD SUBSYSTEM 3-10

3.7. LINKER SUBSYSTEM 3-12

3.8. DOOR.TO.DOOR SUBSYSTEM 3-13

3.9. REGION SUBSYSTEM 3-15

3.10. RESULTS PRINTING SUBSYSTEM 3-17

1. INTRODUCTION

1.1 NOTES ON THE CONTENTS

The variables, modules and data structures used in SMART are identified in two different ways. The first is a title which is unrestricted in length, and which may contain capital letters, periods and question marks. This title is chosen to reflect the contents of the variable or data structure or the function of the module. Thus HAVE.DEMAND.ALREADY? is a logical variable which, if set to .TRUE., indicates that demand matrices have been generated and do not need to be recomputed. DOOR.TO.DOOR.RUN is a subroutine that performs a door-to-door analysis between two zones based on a user card request. FORTRAN, however, cannot accept names of this type, so abbreviatons are usually needed in the program. The actual FORTRAN name for the variable HAVE.DEMAND.ALREADY? is HDEMND and for the subroutine DOOR.TO.DOOR.RUN is DORRUN. In this document, both names will usually be given to identify the item, first the title and then, in parentheses, the FORTRAN name; for example, FEEDER.HEADINGS (FDRHED). When the title and FORTRAN name are the same (as for the data structure DEMAND) the FORTRAN name will not be repeated in parentheses. Occasionally only the FORTRAN name is given.

1.2 SYSTEM SUMMARY

The following analytical routines are the core of the program:

1. FEEDER block;
2. LINKER block;
3. DUMPER block;
4. CBD block;
5. DOOR.TO.DOOR (DOOR) block; and
6. REGIONWIDE (REGION) block.

To supplement these, there are several supporting blocks:

7. The urban structure operations, including modules which load the default network, modify the structure based on card input, and generate basic data such as shortest routes when needed.

8. Travel demand modules load and adjust demand parameters, create demand through generation or reading a file, and alter demand data in response to overriding user specifications.

9. Mode maintenance routines select travel modes for analysis and alter selected or default parameters.

10. Print results.

Overall program control is maintained by SMART.MAIN.PROGRAM (SMARTAJC). This module is the main program, and operates primarily by repetitively reading a keyword card from the input stream and then calling the appropriate subroutine to handle it. In some instances, where the task to be performed is relatively simple, SMART.MAIN.PROGRAM (SMARTAJC) itself performs the requested operation, such as altering a model parameter.

Each analysis block operates in the following manner:

1. All information on the keyword card is checked for completeness and correctness; analysis is suppressed if errors have been encountered, either in this card or in some preceeding card.

2. Appropriate transport modes are selected by default if they have not not already been selected.

3. The selected transport modes are modeled and performance measures are calculated;

4. Results are combined by direction and printed.

Generally, card checking and result reporting are performed by one module, mode selection by another routine, and modal analysis by several modules, one for each distinct transport mode.

2. DATA STRUCTURES

SMART uses 17 common blocks to control its data. This chapter gives the details on these data structures, and on the standard indexes used in SMART. The common blocks may be catagorized as follows:

1. Urban Structure:

   a) URBAN.DEFAULT (UDEFLT) - default structure data

   b) URBAN.STRUCTURE (USTRUC) - structure description

   c) SHORTEST.ROUTES (ROUTES) - node-to-node routes

2. Travel modes:

   a) MODE.DEFINITIONS (MODDEF) - mode type descriptions

   b) MODES.SELECTED (MODSEL) - selected mode descriptions

3. Demand:

   a) DEMAND.PARAMETERS (DPARM) - demand generator controls

   b) DEMAND - actual demand on system

4. Analysis options - These structures are used mostly to pass information from the controlling RUN routine to the results printing HEADING routine. They are:

   a) FEEDER.PARAMETERS (FDRPRM)

   b) LINKER.PARAMETERS (LKRPRM)

   c) DUMPER.PARAMETERS (DPRPRM)

   d) CBD.PARAMETERS (CBDPRM)

   e) DOOR.TO.DOOR.PARAMETERS (DORPRM)

5. Other data structures:

   a) INPUT.OUTPUT.CONTOLS (IOCON) - contains I/O units, etc.

   b) INPUT.RECORD (INREC) - contains the keyword and fields from the current keyword control card.

c) FIXED - named program constants which cannot be changed by the user.

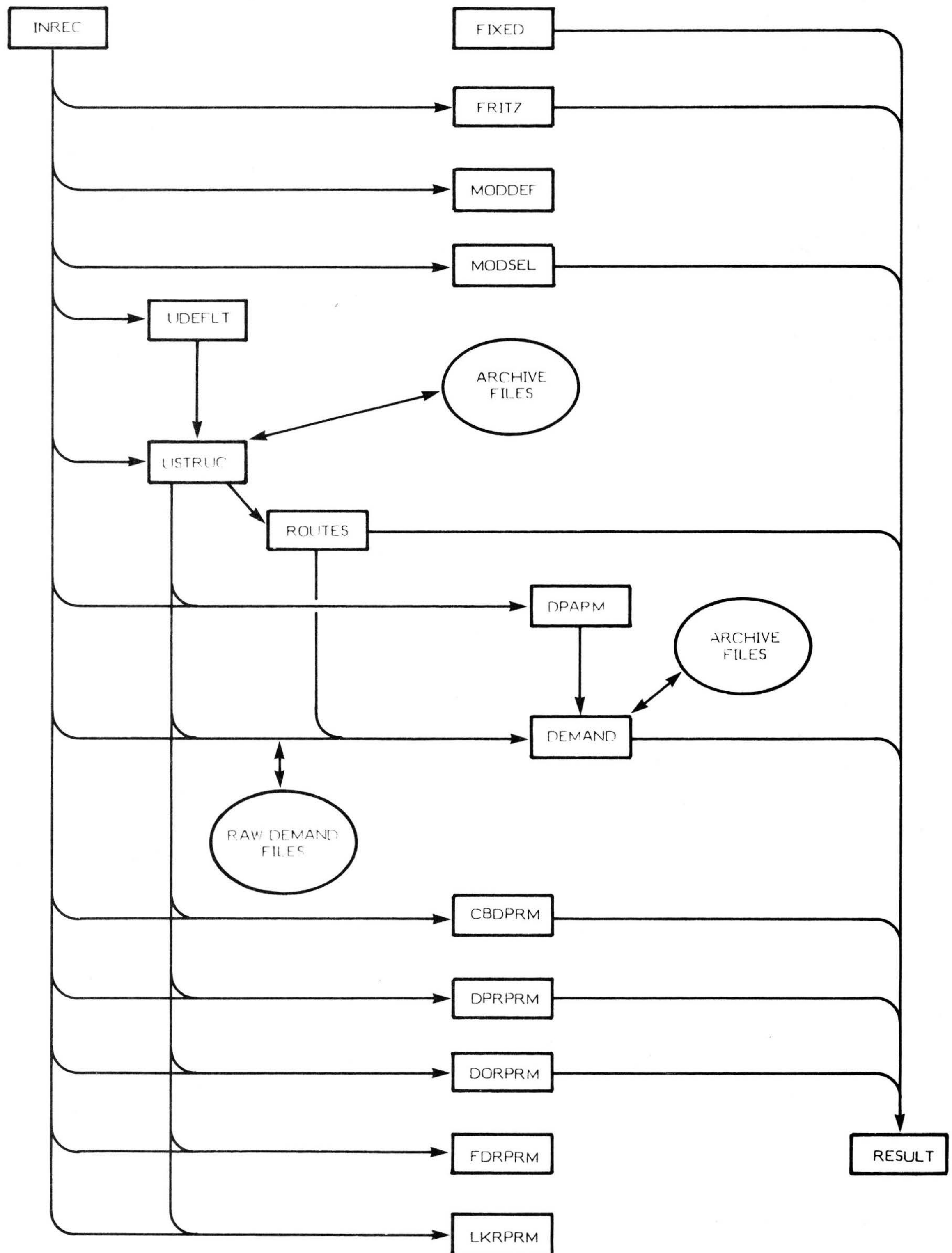
d) FRITZ - miscellaneous parameters, such as transit mode shares.

e) RESULT - model results

Exhibit 2.1 shows the major paths that information takes through the data structures of SMART. For example, INPUT.RECORD (INREC) can affect URBAN.DEFAULT (UDEFLT). URBAN.DEFAULT (UDEFLT) does not directly influence RESULTS (RESULT). However, its current values are used in filling in the URBAN.STRUCTURE (USTRUC) block. The values in URBAN.STRUCTURE (USTRUC) are basically what is written into the archive files when a PUTNET card is encountered. As you can see from the diagram, the urban structure description affects almost everything in SMART.

EXHIBIT 2.1

PRINCIPAL INFORMATION PATHS



2.1 CBD.PARAMETERS (CBDPRM)

This data structure contains the following parameters for the CBD analysis:

CBD.ZONE (CBDZON) (integer)

WALK.DISTANCE.FOR.AUTO (CWDATO) (real)
WALK.DISTANCE.FOR.CARPOOL (CWDCPL) (real)

The FORTRAN COMMON block definition is:

```
C               DATA STRUCTURE CBDPRM AS OF 07/12/79
      COMMON / CBDPRM / CBDZON, CWDATO, CWDCPL
      REAL CWDATO, CWDCPL
      INTEGER CBDZON
C                     DATA STRUCTURE CBDPRM END
```

2.2 DEMAND

This data structure contains the system demand which SMART will model. All the demand-producing code exists only to fill DEMAND. The DEMAND data structure contains:

```
HAVE.DEMAND.ALREADY? (HDEMND) (logical)

For each time period,

    For each direction (1=out,2=in),
        For each zone,
            RESIDENTIAL.VOLUME (VRES(TIME,DIR,ZONE)) (real)
            ACTIVITY.CENTER.VOLUME (VACT(TIME,DIR,ZONE)) (real)

    For each origin node,
        For each destination node,
            TRIP.VOLUME (ODM(TIME,NODE1,NODE2)) (real)

    For each Direction,
        on each link,
            AUTO.TRAFFIC.VOLUME (TRAFIC(TIME,DIR,LINK)) (real)
                      (assuming 0% transit mode share)

            For each Linker mode catagory,
                      (assuming 100% transit mode share)
                TRANSIT.TRAFFIC.VOLUME (TRANLD(TIME,DIR,LINK,MCAT)) (real)
                TRANSIT.BOARDINGS (TRANON(TIME,DIR,LINK,MCAT)) (real)
                TRANSIT.DROPOFFS (TRANOF(TIME,DIR,LINK,MCAT)) (real)

    TOTAL.SYSTEM.DEMAND.RATE (SYSDEM(TIME)) (real)

Note that all volumes are in units of trips per minute.
```

The FORTRAN COMMON block definition is:

```
C               DATA STRUCTURE DEMAND AS OF 07/27/79
      COMMON / DEMAND / HDEMND, VRES(3,2,100), VACT(3,2,100),
     1    ODM(3,50,50), TRAFIC(3,2,100), TRANLD(3,2,100,3),
     2    TRANON(3,2,100,3), TRANOF(3,2,100,3), SYSDEM(3), HTRLOD,
     3    IGDEM(3)
      REAL VRES, VACT, ODM, TRAFIC, TRANLD, TRANON, TRANOF, SYSDEM,
     1    IGDEM
      LOGICAL HDEMND,HTRLOD
      REAL GDEM(14708)
      EQUIVALENCE (HDEMND,GDEM(1))
C     GDEM IS USED ONLY BY PUTNET AND GETNET
C                       DATA STRUCTURE DEMAND END
```

2.3 DOOR.TO.DOOR.PARAMETERS (DORPRM)

This structure contains the parameters needed for a card-initiated door-to-door analysis. It contains:

DOOR.TO.DOOR.ORIGIN.ZONE (ORGZON) (integer)
DOOR.TO.DOOR.DESTINATION.ZONE (DESZON) (integer)

FEEDER.ORIGIN.ZONE? (FEDORG) (logical)
FEEDER.DESTINATION.ZONE? (FEDDES) (logical)

For each zone,
- DOOR.TO.DOOR.CBD.MODE (DCMODE(ZONE)) (integer)
- DOOR.TO.DOOR.FEEDER.MODE (DFMODE(ZONE)) (integer)
- DOOR.TO.DOOR.DUMPER.MODE (DDMODE(ZONE)) (integer)

LINKER.MODE.CATEGORY (LCAT) (integer)

The FORTRAN COMMON block definition is:

```
C                 DATA STRUCTURE DORPRM AS OF 09/06/79
      COMMON / DORPRM / ORGZON, DESZON, DCMODE(100), DFMODE(100),
     1    DDMODE(100), LCAT, FEDDES, FEDORG
      INTEGER ORGZON, DESZON, DCMODE, DFMODE, DDMODE,
     2    LCAT
      LOGICAL FEDDES, FEDORG
C                    DATA STRUCTURE DORPRM END
```

2.4 DEMAND.PARAMETERS (DPARM)

This structure provides storage for demand generation parameters. It was designed to accomodate a wide variety of demand generation procedures. Thus, while the default procedure may use these vectors for particular purposes, there is no real limit on how another generation procedure may use these same vectors. This structure contains:

```
HAVE.DEMAND.PARAMETERS? (HDPARM) (logical)

For each parameter (up to 5),
   for each zone,
      ZONAL.DEMAND.PARAMETER (ZDPARM(1-5,ZONE)) (real)
            Default methodology uses:
                1 as population density
                2 as employment density
                3 as average commute length
                4 as average non-commute home-based trip length
                5 as average non-home based work trip length

For each parameter,
   For each ring,
      DEFAULT.ZONAL.DEMAND.PARAMETER (DZDPRM(1-5,RING)) (real)

For each time period
   For each direction,
      For each trip type (up to 3),
         TIME.PROPORTION (TPROP(TIME,DIR,TTYPE)) (real)

EXTRA.DEMAND.PARAMETERS (DXPARM(1-8)) (real)
         Default methodology uses:
             1 as commute trips per day per unit population.
             2 as non-commute home-based trips per day
                   per unit population.
             3 as non-home based work trips per day
                   per unit employment.
```

The FORTRAN COMMON block definition is:

```
C               DATA STRUCTURE DPARM AS OF 03/22/79
      COMMON / DPARM / HDPARM, ZDPARM(5,100), DZDPRM(5,8),
     1    TPROP(4,2,3), DXPARM(8)
      REAL ZDPARM, DZDPRM, TPROP, DXPARM
      LOGICAL HDPARM
C                     DATA STRUCTURE DPARM END
```

2.5 DUMPER.PARAMETERS (DPRPRM)

This data structure contains the parameters needed to run a card-initiated DUMPER analysis:

DUMPER.ZONE (DPRZON) (integer)

DUMPER.TRANSIT.FRACTION (DPRTF) (real)
which is the fraction of transit trips desiring collection or distribution by DUMPER.

WALK.DISTANCE.FOR.AUTO (DWDATO) (real)
WALK.DISTANCE.FOR.CARPOOL (DWDCPL) (real)
WALK.DISTANCE.FOR.FIXED.ROUTE.BUS (DWDFRB) (real)

The FORTRAN COMMON block definition is:

```
C               DATA STRUCTURE DPRPRM AS OF 07/19/79
      COMMON /DPRPRM /DPRZON,DPRTF,DWDFRB,DWDATO, DWDCPL
      REAL DPRTF,DWDFRB,DWDATO,DWDCPL
      INTEGER DPRZON
C                  DATA STRUCTURE DPRPRM END
```

2.6 FEEDER.PARAMETERS (FDRPRM)

This data structure contains the parameters for a card-initiated FEEDER analysis:

FEEDER.ZONE (FDRZON) (integer)
FEEDER.ANALYSIS.TYPE (FDRATP) (integer)
FEEDER.DENSITY.RATIO (FDRDRA) (real)
FEEDER.SUBAREA.RATIO (FDRARA) (real)

The FORTRAN COMMON block definition is:

```
C                 DATA STRUCTURE FDRPRM AS OF 04/12/79
      COMMON / FDRPRM / FDRZON, FDRATP, FDRDRA, FDRARA
      REAL FDRDRA, FDRARA
      INTEGER FDRZON, FDRATP
C                       DATA STRUCTURE FDRPRM END
```

2.7 FIXED

This data structure contains certain items which might have been coded as constants in the SMART program. Because, at the time of writing SMART, there was some question about the exact values to be used, we coded these items as variables that are initialized in BLOCK DATA and never changed. This allows one to modify them without changing dozens of lines of program code. Nonetheless, these items are not as easily changed as are common model parameters. They are, in some sense, FIXED parameters. This structure includes:

ALPHA (real)
BETA.OVER.TWO (BETAO2) (real)

LOCAL.DISTANCE.FRACTION (LOCDIS) (real)
LOCAL.DISTANCE.FRACTION.FOR.CBD (LOCDIC) (real)

AUTO.TRIPS.PER.DAY (ATPDAY) (real)

FRACTION.OF.USEABLE.ROAD (FROAD) (real)

The FORTRAN COMMON block definition is:

```
C               DATA STRUCTURE FIXED AS OF 07/12/79
      COMMON / FIXED / ALPHA, BETAO2, LOCDIS, ATPDAY, LOCDIC, FROAD
      REAL ALPHA, BETAO2, LOCDIS, ATPDAY, LOCDIC, FROAD
C                       DATA STRUCTURE FIXED END
```

When waiting times are somewhat controlled, but still contain random factors (as for instance coordinated transfer times) SMART uses the formula:

wait time = alpha + beta * headway / 2.

where alpha = 1.71 minutes and beta = 0.57. These terms are incorporated into ALPHA and BETA.OVER.TWO (BETAO2).

SMART assumes that the average length of a residential trip in a zone, from the residential location to the point where traffic enters and leaves the network, is 0.67 times the length of a side of the square zone. Although a strictly uniform rectiliniar model would use 0.75 times the zone side length, 0.67 was chosen to reflect the tendency of activity to collect near the network. This factor is found in LOCDIS and LOCDIC.

In order for SMART to allocate the capital cost of an automobile properly, it must make some assumption about how many trips the average car makes in a day. AUTO.TRIPS.PER.DAY (ATPDAY) is the assumed number of one-way trips the average auto makes per day, and is currently 2.3.

Smead's equation for CBD congestion makes certain assumptions about road width, auto parking areas, etc., which determine the fraction of

total land which is taken up by traffic lanes. This fraction is in FRACTION.OF.USEABLE.ROAD (FROAD).[1]

---

[1] Smeed, R.J., "Traffic Studies and Urban Congestion," Journal of Transport Economics and Policy, January 1968.

2.8 FRITZ

This data structure contains miscellaneous items which somehow didn't quite fit anywhere else. The name reflects the structure's lack of central theme. It contains:

```
For each time period,
   TIME.PERIOD.LENGTH (TIMLEN(TIME)) (real)

NUMBER.OF.MODE.SHARES (NMSI) (integer)

For each time period,
   for each mode share index,
      TRANSIT.MODE.SHARE (TMS(TIME,MSI)) (real)

CARPOOL.FRACTION (CPFRAC) (real)
CARPOOL.TIME.FACTOR (CPTIME) (real)

WALKING.SPEED (WSPEED) (real)
```

The FORTRAN COMMON block definition is:

```
C               DATA STRUCTURE FRITZ AS OF 06/26/79
      COMMON / FRITZ / TIMLEN(4), NMSI, TMS(3,6), CPFRAC, CPTIME,
     1    WSPEED
      REAL TIMLEN, TMS, CPFRAC, CPTIME, WSPEED
      INTEGER NMSI
C                           DATA STRUCTURE FRITZ END
```

2.9 INPUT.RECORD (INREC)

This data structure contains the fields from the latest keyword card read (except for columns 73/80, which are read only for printing). It contains:

KEYWORD (KEYWRD) (character)
CARD.FIELDS (FIELDS(1-8)) (character)

The FORTRAN COMMON block definition is:

```
C               DATA STRUCTURE INREC AS OF 03/16/79
      COMMON / INREC / KEYWRD, FIELDS(8)
      DOUBLE PRECISION KEYWRD, FIELDS
C                     DATA STRUCTURE INREC END
```

2.10 INPUT.OUTPUT.CONTROLS (IOCON)

This data structure contains information related to input/output, principally I/O units and report format controls. The count of the number of errors encountered is in this data structure because any module detecting an error must both count it and print a diagnostic message. The contents are:

CARD.INPUT.UNIT (CARDUN) (integer)
PRINTING.UNIT (PRNTUN) (integer)
GET.NETWORK.UNIT (GNETUN) (integer)
PUT.NETWORK.UNIT (PNETUN) (integer)
READ.DEMANDS.UNIT (RDEMUN) (integer)
WRITE.DEMANDS.UNIT (WDEMUN) (integer)

NUMBER.OF.LINES.PER.PAGE (LPP) (integer)
NUMBER.OF.LINES.THIS.PAGE (LINES) (integer)
CURRENT.PAGE.NUMBER (PAGENO) (integer)
REPORT.TITLE (TITLE(1-8)) (character)
REPORT.SUBTITLE (STITLE(1-8)) (character)
TODAYS.DATE (DATE) (character)

NUMBER.OF.ERRORS (ERRORS) (integer)

The FORTRAN COMMON block definition is:

```
C                 DATA STRUCTURE IOCON OF 04/14/80
      COMMON / IOCON / TITLE(8), STITLE(8), DATE, CARDUN, PRNTUN,
     1    GNETUN, PNETUN, LPP, LINES, PAGENO, ERRORS, RDEMUN,
     2    WDEMUN
      DOUBLE PRECISION TITLE, STITLE, DATE
      INTEGER CARDUN, PRNTUN, GNETUN, PNETUN, LPP, LINES, PAGENO,
     1    ERRORS, RDEMUN, WDEMUN
C                     DATA STRUCTURE IOCON END
```

It is quite likely that, when installing SMART on a particular computer, the initialization for the input/output units will have to be changed. They are presently set to CARDUN = 5 and PRNTUN = 6, which is normal but by no means universal.

2.11 LINKER.PARAMETERS (LKRPRM)

This data structure holds any parameters needed for a card-initiated LINKER analysis. It contains:

ORIGIN.NODE (ONODE) (integer)
DESTINATION.NODE (DNODE) (integer)

The FORTRAN COMMON block definition is:

```
C               DATA STRUCTURE LKRPRM AS OF 06/14/79
      COMMON / LKRPRM / ONODE, DNODE
      INTEGER ONODE, DNODE
C                    DATA STRUCTURE LKRPRM END
```

2.12 MODE.DEFINITIONS (MODDEF)

This data structure defines the types of modes that SMART can model, including default parameter values. It contains:

```
NUMBER.OF.DEFINED.MODES (NDMOD) (integer)

For each defined mode,
   TYPE.IDENTIFIER (DMIDNT(DMOD)) (character)
   MODE.LABEL (DMLABL(1-4,DMOD)) (character)
   FEEDER.MODE? (DMFEDR(DMOD)) (logical)
   LINKER.MODE? (DMLNKR(DMOD)) (logical)
   DUMPER.MODE? (DMDMPR(DMOD)) (logical)
   CBD.MODE? (DMCBD(DMOD)) (logical)

   COST (DMC(DMOD)) (real)
   COST.PER.VEHICLE (DMCV(DMOD)) (real)
   COST.PER.VEHICLE.MINUTE (DMCVH(DMOD)) (real)
   COST.PER.VEHICLE.MILE (DMCVM(DMOD)) (real)
   COST.PER.ROUTE.MILE (DMCRM(DMOD)) (real)

   UNCONGESTED.LOCAL.SPEED (DMSPED(DMOD)) (real)
   MINIMAL.STOP.TIME (DMSTPT(DMOD)) (real)
   ADDITIONAL.STOP.TIME.PER.PASSENGER (DMSTPP(DMOD)) (real)

   MILES.PER.GALLON.AT.LOCAL.SPEED (DMMPG(DMOD)) (real)
   MPG.CURVE.SHAPE.PARAMETER (DMMPGX(DMOD)) (real) (unused)

   VEHICLE.CAPACITY.OR.LOADING (DMVCAP(DMOD)) (real)

   MINIMUM.HEADWAY (DMHWMN(DMOD)) (real)
   MAXIMUM.HEADWAY (DMHWMX(DMOD)) (real)
   MAXIMUM.ROUTE.SPACING (DMRSMX(DMOD)) (real)
   STOP.SPACING (DMSTPS(DMOD)) (real)

   CAR.EQUIVALENTS.PER.VEHICLE (DMCRPV(DMOD)) (real)

   FRACTION.WHO.WALK (DMFWLK(DMOD)) (real) (unused)
   FRACTION.WHO.PARK (DMFPRK(DMOD)) (real) (unused)

   For each time period,
      TRAIN.SIZE (DMTRAN(TIME,MOD)) (real)
      SUBSCRIPTION.SERVICE? (DMSUB(TIME,MOD)) (logical)

   TRANSFER.COORDINATION (DMTCRD(DMOD)) (integer)
         whose codes signify the following wait times:
         1 = RANDOM (headway/2)
         2 = TIMED (alpha+beta*headway/2)
         3 = THROUGH (no wait time)
```

The FORTRAN COMMON block definition is:

```
C               DATA STRUCTURE MODDEF AS OF 06/26/79
      COMMON / MODDEF / DMIDNT(8), DMLABL(4,8), NDMOD, DMFEDR(8),
     1    DMLNKR(8), DMDMPR(8), DMCBD(8), DMC(8), DMCV(8),
     2    DMCVH(8), DMCVM(8), DMCRM(8), DMSPED(8), DMSTPT(8),
     3    DMSTPP(8), DMMPG(8), DMMPGX(8), DMVCAP(8),
     4    DMHWMN(8), DMHWMX(8), DMRSMX(8), DMSTPS(8),
     5    DMCRPV(8), DMFWLK(8), DMTRAN(3,8), DMSUB(3,8), DMTCRD(8),
     6    DMFPRK(8)
      DOUBLE PRECISION DMIDNT, DMLABL
      REAL DMC, DMCV, DMCVH, DMCVM, DMCRM, DMSPED, DMSTPT, DMSTPP,
     1    DMMPG, DMMPGX, DMVCAP, DMHWMN, DMHWMX, DMRSMX,
     2    DMSTPS, DMCRPV, DMFWLK, DMTRAN, DMFPRK
      INTEGER NDMOD, DMTCRD
      LOGICAL DMFEDR, DMLNKR, DMDMPR, DMCBD, DMSUB
C                    DATA STRUCTURE MODDEF END
```

MPG.CURVE.SHAPE.PARAMETER (DMMPGX) was established to vary fuel mileage with speed. Currently SMART uses a single fuel consumption in miles-per-gallon, without adjustment.

FRACTION.WHO.WALK (DMFWLK) and FRACTION.WHO.PARK (DMFPRK) are leftovers from an abandoned methodology. They were part of the effort to model walking in DUMPER and CBD, and are no longer used. These vectors are unused in that they have the following activity:

1. There is a vector in MODE.DEFINTIONS (MODDEF) and a corresponding vector in MODES.SELECTED (MODSEL).

2. The modal definition vector is initialized in BLOCK DATA.

3. When a mode is selected, the subroutine NEW.MODE (NEWMOD) transfers the value from the modal definition vector to the modal selection vector.

4. PRINT.MODAL.PARAMETERS (PMODES) may or may not print the value.

5. No other module in SMART accesses either of the vectors.

2.13 MODES.SELECTED (MODSEL)

This data structure contains the specifications of the selected modes. It contains:

```
AUTO.MODE.NUMBER (JAUTO) (integer)
CARPOOL.MODE.NUMBER (JCARPL) (integer)
FIXED.ROUTE.BUS.MODE.NUMBER (JFRB) (integer)
FLEXIBLE.ROUTE.BUS.MODE.NUMBER (JFLEX) (integer)
HEAVY.RAIL.MODE.NUMBER (JHRAIL) (integer)
LIGHT.RAIL.MODE.NUMBER (JLRAIL) (integer)
AUTOMATED.GUIDEWAY.MODE.NUMBER (JAGT) (integer)
PRIVATE.AUTO.TRANSIT.MODE.NUMBER (JPAT) (integer)

NUMBER.OF.MODES.SELECTED (NMOD) (integer)
MAXIMUM.NUMBER.OF.MODES.SELECTED (MXMOD) (integer)

For each selected mode,
   MODE.IDENTIFIER (MIDENT(MOD)) (character)
   MODE.TYPE (MTYPE(MOD)) (integer)
         which is one of the above travel mode numbers.
   MODE.LABEL (MLABL(1-4,MOD)) (character)
   MODE.SUBROUTINE (MSUBR(MOD)) (integer)
            iwhose codes are:
               1 = FEEDER
               2 = LINKER
               3 = DUMPER
               4 = CBD
               5 = REGION totalling area
               6 = DOOR.TO.DOOR totalling area

   COST (MC(MOD)) (real)
   COST.PER.VEHICLE (MCV(MOD)) (real)
   COST.PER.VEHICLE.MINUTE (MCVH(MOD)) (real)
   COST.PER.VEHICLE.MILE (MCVM(MOD)) (real)
   COST.PER.ROUTE.MILE (MCRM(MOD)) (real)

   UNCONGESTED.LOCAL.SPEED (MSPEED(MOD)) (real)
   MINIMAL.STOP.TIME (MSTOPT(MOD)) (real)
   ADDITIONAL.STOP.TIME.PER.PASSENGER (MSTOPP(MOD)) (real)

   MILES.PER.GALLON.AT.LOCAL.SPEED (MMPG(MOD)) (real)
   MPG.CURVE.SHAPE.PARAMETER (MMPGX(MOD)) (real) (unused)

   VEHICLE.CAPACITY.OR.LOADING (MVCAP(MOD)) (real)

   MINIMUM.HEADWAY (MHWMIN(MOD)) (real)
   MAXIMUM.HEADWAY (MHWMAX(MOD)) (real)
   MAXIMUM.ROUTE.SPACING (MRSMAX(MOD)) (real)
   STOP.SPACING (MSTOPS(MOD)) (real)

   CAR.EQUIVALENTS.PER.VEHICLE (MCARPV(MOD)) (real)

   FRACTION.WHO.WALK (MFWALK(MOD)) (real) (unused)
```

```
      FRACTION.WHO.PARK (MFPARK(MOD)) (real) (unused)

      For each time period,
         TRAIN.SIZE (MTRAIN(TIME,MOD)) (real)
         SUBSCRIPTION.SERVICE? (MSUB(TIME,MOD)) (logical)

      SUPPRESS.ANALYSIS? (MNORUN(MOD)) (logical)

      TRANSFER.COORDINATION, (MTCORD(MOD)) (integer)
            whose codes (and wait times) are:
               1 = RANDOM (headway/2)
               2 = TIMED (alpha+beta*headway/2)
               3 = THROUGH (no wait time)

   For each primary analysis type (FEEDER, LINKER, DUMPER, and CBD),
      SELECT.DEFAULTS? (SMDSEL(1-4)) (logical)

   For each zone,
      DESIGNATED.FEEDER.MODE (ZFMODE(ZONE)) (integer)
      DESIGNATED.DUMPER.MODE (ZDMODE(ZONE)) (integer)
      DESIGNATED.CBD.MODE (ZCMODE(ZONE)) (integer)
```

The FORTRAN COMMON block definition is:

```
C               DATA STRUCTURE MODSEL AS OF 06/26/79
      COMMON / MODSEL / MIDENT(25), MLABL(4,25), NMOD, MXMOD,
     1    MTYPE(25), MSUBR(25), MC(25), MCV(25), MCVH(25), MCVM(25),
     2    MCRM(25), MSPEED(25), MSTOPT(25), MSTOPP(25), MMPG(25),
     3    MMPGX(25), MVCAP(25), MHWMIN(25), MHWMAX(25),
     4    MRSMAX(25), MSTOPS(25), MCARPV(25), MFWALK(25),
     5    MTRAIN(3,25), SMDSEL(4), MNORUN(25), MSUB(3,25),
     6    MTCORD(25), MFPARK(25), JAUTO, JCARPL, JFRB, JFLEX,
     7    JHRAIL, JLRAIL, JAGT, JPAT, ZFMODE(100), ZDMODE(100),
     8    ZCMODE(100)
      DOUBLE PRECISION MIDENT, MLABL
      REAL MC, MCV, MCVH, MCVM, MCRM, MSPEED, MSTOPT, MSTOPP, MMPG,
     1    MMPGX, MVCAP, MHWMIN, MHWMAX, MRSMAX, MSTOPS,
     2    MCARPV, MFWALK, MTRAIN, MFPARK
      INTEGER NMOD, MXMOD, MTYPE, MSUBR, MTCORD, JAUTO, JCARPL,
     1    JFRB, JFLEX, JHRAIL, JLRAIL, JAGT, JPAT, ZFMODE, ZDMODE,
     2    ZCMODE
      LOGICAL SMDSEL, MNORUN, MSUB
C                    DATA STRUCTURE MODSEL END
```

The variables <travel>.MODE.NUMBER (J----) do double duty. First, they are the codes found in the vector MODE.TYPE (MTYPE). Second, they are indexes into the vectors in MODE.DEFINITIONS (MODDEF). All modules use these variables to decode mode types, thereby making it easier to add new mode types, to delete mode types, or to re-order existing mode types (for which there should be no need, since no code depends upon the relative ordering of the mode types).

Please see the discussion of the MODE.DEFINITIONS (MODDEF) data structure for an explanation of the unused vectors.

SUPPRESS.ANALYSIS (MNORUN) is required because there are situations where not all selected modes can be modelled at the same time. SMART needed a mechanism to shut off selected modes and turn them back on at will. For example, in a regionwide analysis, FEEDER will be called to model service in each zone, using the designated FEEDER service type for that zone. During this analysis, other selected FEEDER service types must be deactivated.

SELECT.DEFAULTS (SMDSEL) is the mechanism by which the user can prevent default mode selection when he/she wants to model only certain specifically selected modes. Because it only prevents selection of the defaults, it has no effect if the defaults have already been selected. SMART currently provides no method by which the user can turn default selection back on.

This data structure also contains the vectors that give the designated FEEDER, DUMPER, and CBD modes for each zone. The vectors contain values which are indexes into the rest of the MODSEL structure. Because they are indexed by ZONE, these vectors constitute the only interconnection between urban structure and modes selected.

2.14 RESULTS (RESULT)

All the computational effort in SMART exists to produce values in this data structure. It contains:

```
For each time period,
   for each direction, *
      for each transit mode share alternative,
         for each selected mode,

            COST.PER.TRIP (COST(TIME,DIR,MSI,MOD)) (real)
            TRAVEL.TIME (TTIME(TIME,DIR,MSI,MOD)) (real)
            TRAVEL.TIME.STD.DEV (TTIMED(TIME,DIR,MSI,MOD)) (real)
            FUEL.CONSUMPTION.PER.TRIP (FUEL(TIME,DIR,MSI,MOD)) (real)
            FLEET.SIZE (FLEET(TIME,DIR,MSI,MOD)) (real)
            HEADWAY (HEADWY(TIME,DIR,MSI,MOD)) (real)
            AVERAGE.LOADING (LOADNG(TIME,DIR,MSI,MOD)) (real)
            VEHICLE.MILES.PER.MINUTE (VMPM(TIME,DIR,MSI,MOD)) (real)
            PICKUPS.PER.MINUTE (VPPM(TIME,DIR,MSI,MOD)) (real)

         LINKER.EXPRESS.BUS? (EXPRES(TIME,DIR,MSI)) (logical)
```

* For FEEDER, DUMPER, and CBD modes, direction 1 is outbound and direction 2 is inbound.
For LINKER modes, direction 1 is for the selected trip, and direction 2 is for the reverse trip.
For all modes, direction 3 is an accumulation area used by the driver routine to sum findings accross both directions or, for REGION, accross the entire system.

The FORTRAN COMMON block definition is:

```
C              DATA STRUCTURE RESULT AS OF 07/12/79
      COMMON / RESULT / COST(3,3,6,25), TTIME(3,3,6,25),
     1    TTIMED(3,3,6,25), FUEL(3,3,6,25), FLEET(3,3,6,25),
     2    HEADWY(3,3,6,25), LOADNG(3,3,6,25), VMPM(3,3,6,25),
     3    VPPM(3,3,6,25), EXPRES(3,3,6)
      REAL COST, TTIME, TTIMED, FUEL, FLEET, HEADWY, LOADNG, VMPM,
     1    VPPM
      LOGICAL EXPRES
C                    DATA STRUCTURE RESULT END
```

2.15 SHORTEST.ROUTES (ROUTES)

This data structure holds the routes which lead from one network node to another. It contains:

```
HAVE.SHORTEST.ROUTES.ALREADY? (HROUTS) (logical)

For each origin node,
   For each destination node,
      LAST.AUTO.ROUTE.LINK (ROUTL(NODE2,NODE1)) (integer)
      AUTO.ROUTE.DISTANCE (ROUTD(NODE2,NODE1)) (real)

NUMBER.OF.TRANSIT.ROUTE.LEGS (NLEG) (integer)
For each transit route leg,
   TRANSIT.ROUTE.LINK (TLINK(LEG)) (integer)
   TRANSIT.ROUTE.DIRECTION (TDIR(LEG)) (integer)
   TRANSIT.ROUTE.MODE.CATAGORY (TMCAT(LEG)) (integer)
```

The FORTRAN COMMON block definition is:

```
C              DATA STRUCTURE ROUTES AS OF 06/18/79
      COMMON / ROUTES / HROUTS, ROUTL(50,50), ROUTD(50,50), NLEG,
     1    TLINK(100), TDIR(100), TMCAT(100)
      REAL ROUTD
      INTEGER ROUTL, NLEG, TLINK, TDIR, TMCAT
      LOGICAL HROUTS
C                    DATA STRUCTURE ROUTES END
```

The logical switch HAVE.SHORTEST.ROUTES.ALREADY? (HROUTS) refers to the automobile shortest routes only. Transit routes are re-calculated whenever one is needed.

Auto shortest routes are encoded in the matrix LAST.AUTO.ROUTE.LINK (ROUTL). Note that the destination is the first index, and the origin the second. Although this seems backwards, it arises because our shortest routes routine operates on one origin at a time. The resultant vector is one column of this matrix (FORTRAN stores matrices by columns). For a given origin, the routes to the destinations form a tree, and the segments of that tree are stored in ROUTL. Thus one can only trace a route backwards. It is done with the following algorithm:

1. set current node to destination node.

2. if current node is origin node, terminate; otherwise:

3. obtain link number from LAST.AUTO.ROUTE.LINK (ROUTL).

4. next node number is whichever endpoint of the current link does not match the current node.

5. do whatever needs done for this link

6. set current node to next node, and go back to step 2.

Although this technique of handling shortest routes may seem strange, it requires considerably less memory than other methods. Because data requirements are order(N**2), (Order(N**3) in some alternate techniques), reducing the size of shortest route storage was a major concern.

Unlike auto shortest routes, only one transit route is stored at a time. Hence any transit route needed is generated when it is required. By definition the transit route can extend from one node to the same node -- in this case, the number of transit route legs (NLEG) is zero.

2.16 URBAN.DEFAULT (UDEFLT)

This structure contains most of the information needed to generate a ring/corridor network. The contents are:

```
MAXIMUM.NUMBER.OF.RINGS (MXRING) (integer)
For each ring,
   RING.OUTER.RADIUS (RADIUS(RING)) (real)
   DEFAULT.OUTER.RADIUS (DRADUS(RING)) (real)
   DEFAULT.PARKING.COST (DZPARK(RING)) (real)

MAXIMUM.NUMBER.OF.CORRIDORS (MXCOR) (integer)
For each corridor,
   CORRIDOR.ANGLE (ANGLE(COR)) (real)

For each possible default link,
   DEFAULT.LINK.IDENTIFIER (DLIDNT(DLINK)) (character)

For each possible default node,
   DEFAULT.NODE.IDENTIFIER (DNIDNT(DNODE)) (character)

For each possible default zone,
   DEFAULT.ZONE.IDENTIFIER (DZIDNT(DZONE)) (character)

For radial (1) and circumferential (2) highways,
   DEFAULT.LINK.TYPE(DLTYPE(1-2)) (integer)
   DEFAULT.NUMBER.OF.LANES (DLLANS(1-2)) (real)
   DEFAULT.DIAMOND.LANE? (DLDMND(1-2)) (logical)
   DEFAULT.LIGHT.RAIL? (DLLRAL(1-2)) (logical)
   DEFAULT.HEAVY.RAIL? (DLHRAL(1-2)) (logical)
   DEFAULT.SPEED (DLSPED(1-2)) (real)
   DEFAULT.LIGHT.RAIL.SPEED (DLLSPD(1-2)) (real)
```

The FORTRAN COMMON block definition is:

```
C               DATA STRUCTURE UDEFLT AS OF 05/19/79
      COMMON / UDEFLT / DLIDNT(100), DNIDNT(50), DZIDNT(100),
     1    MXRING, RADIUS(8), MXCOR, ANGLE(6),
     2    DLTYPE(2), DLLANS(2), DLDMND(2), DLLRAL(2), DLHRAL(2),
     3    DZPARK(8), DRADUS(8), DLSPED(2), DLLSPD(2)
      DOUBLE PRECISION DLIDNT, DNIDNT, DZIDNT
      REAL RADIUS, ANGLE, DLLANS, DZPARK, DRADUS, DLSPED, DLLSPD
      INTEGER MXRING, MXCOR, DLTYPE
      LOGICAL DLDMND, DLLRAL, DLHRAL
C                     DATA STRUCTURE UDEFLT END
```

Notice that the actual number of rings and corridors is missing. Even after the network is generated, LINKSET, ZONESET, and ZDPSET cards need these values to translate ring and corridor numbers into link or zone indexes. Consequently, NRING and NCOR must be archived with the network. All the values in URBAN.DEFAULT (UDEFLT) are used only at that instant whan the default network is generated.

Both vectors for ring radius must be initialized to the same default values.

The default link, node, and zone identifiers are stored in the vectors as they would be in the actual identifier arrays for a full ring/corridor system (with the maximum number of each).

2.17 URBAN.STRUCTURE (USTRUC)

This data structure defines the geographic characteristics of the system being modelled. It contains:

```
HAVE.URBAN.STRUCTURE.ALREADY? (HUSTRC) (logical)

URBAN.STRUCTURE.TYPE (UTYPE) (integer)
          whose codes are:
          1 = ring/corridor system
          2 = arbitrary network

If this is a ring/corridor structure,
   NUMBER.OF.RINGS (NRING) (integer)
   NUMBER.OF.CORRIDORS (NCOR) (integer)

MAXIMUM.NUMBER.OF.LINKS (MXLINK) (integer)
NUMBER.OF.LINKS (NLINK) (integer)
For each LINK,
   LINK.IDENTIFIER (LIDENT(LINK)) (character)
   LINK.TYPE (LTYPE(LINK)) (integer)
            whose codes are:
            0 = missing
            1 = freeway
            2 = arterial
   NUMBER.OF.LANES (LLANES(LINK)) (real)
   DIAMOND.LANE? (LDMND(LINK)) (logical)
   LIGHT.RAIL? (LLRAIL(LINK)) (logical)
   HEAVY.RAIL? (LHRAIL(LINK)) (logical)
   LENGTH (LLENTH(LINK)) (real)
   UNCONGESTED.AUTO.SPEED (LSPEED(LINK)) (real)
   LIGHT.RAIL.SPEED (LLSPED(LINK)) (real)
   For each end of the link,
      NODE.AT.END (LNODE(1-2,LINK)) (integer)

For each non-null link type (1=freeway,2=arterial),
   CAPACITY.PER.LANE (CAPPLN(1-2)) (real)

MAXIMUM.NUMBER.OF.NODES (MXNODE) (integer)
NUMBER.OF.NODES (NNODE) (integer)
For each node,
   NODE.IDENTIFIER (NIDENT(NODE)) (character)
   LINK.CONNECTIONS.LIST.START (NODPP(NODE)) (integer)
   ZONE.CONNECTIONS.LIST.START (NODZPP(NODE)) (integer)
   For each entry in the links connections list,
      LINK.CONNECTED.TO (NODP(1,IPP)) (integer)
      NODE.CONNECTED.TO (NODP(2,IPP)) (integer)
   For each entry in the zones connections list,
      ZONE.CONNECTED.TO (NODZP(IPP)) (integer)

MAXIMUM.NUMBER.OF.ZONES (MXZONE) (integer)
NUMBER.OF.ZONES (NZONE) (integer)
For each zone,
   ZONE.INDENTIFIER (ZIDENT(ZONE)) (character)
```

```
NODE.CONNECTED.TO (ZNODE(ZONE)) (integer)
ZONE.TYPE (ZTYPE(ZONE)) (integer)
            whose codes are:
               1 = CBD
               2 = non-CBD
               3 = zone to ignore
ZONE.SIZE (ZSIZE(ZONE)) (real)
ZONE.SIDE.LENGTH (ZSIDE(ZONE)) (real)
ZONE.PARKING.COST (ZPARK(ZONE)) (real)
```

The FORTRAN COMMON block definition is:

```
C                DATA STRUCTURE USTRUC AS OF 05/19/79
      COMMON / USTRUC / LIDENT(100), NIDENT(50), ZIDENT(100),
     1    HUSTRC, MXLINK, NLINK, LTYPE(100), LLANES(100),
     2    LDMND(100), LLRAIL(100), LHRAIL(100), LLENTH(100),
     3    LNODE(2,100), CAPPLN(2), MXNODE, NNODE, NODPP(51),
     4    NODZPP(51), NODP(2,200), NODZP(100), MXZONE,
     5    NZONE, ZNODE(100), ZTYPE(100), ZSIZE(100), ZPARK(100),
     6    UTYPE, ZSIDE(100), NRING, NCOR, LSPEED(100), LLSPED(100)
      DOUBLE PRECISION LIDENT, NIDENT, ZIDENT
      REAL LLANES, LLENTH, CAPPLN, ZSIZE, ZPARK, ZSIDE, LSPEED,
     1    LLSPED
      INTEGER MXLINK, NLINK, LTYPE, LNODE, MXNODE, NNODE, NODPP,
     1    NODZPP, NODP, NODZP, MXZONE, NZONE, ZNODE, ZTYPE, UTYPE,
     2    NRING, NCOR
      LOGICAL HUSTRC, LDMND, LLRAIL, LHRAIL
      REAL GNET(2614)
      EQUIVALENCE (GNET(1),LIDENT(1))
C                     DATA STRUCTURE USTRUC END
```

The method used to encode the interconnections between links and nodes and zones is efficient but not at all obvious. Imagine a vector which contains an entry for each node. That entry is a list of the links which are attached to that node. Such a vector cannot be implemented directly in FORTRAN, since each element of the vector is a list (array). We could have used a matrix where each row represented a node and the columns listed the links. Such a matrix would require as many columns as the maximum number of links attached to any one node. This would result in a lot of wasted space, since most nodes have far fewer than the maximum attached links. We have implemented this links-listing vector by using an array as if it consisted of a fixed number of rows, each containing a variable number of columns. NODP contains a segment for each node, and the segment begins at NODPP(NODE) and goes through NODPP(NODE+1)-1. Each row in this segment of NODP refers to one link attached to this node; NODP(1,?) gives the index of the link, and NODP(2,?) gives the index of the node on the other end of the link. Every link appears twice in NODP, and thus NODP must be dimensioned 2 by 2*MXLINK. To find the segment used, we start at NODPP(NODE) and go through NODPP(NODE+1)-1, even if NODE = MXNODE. Hence NODPP must be dimensioned by MXNODE+1. A similar process uses NODZPP(NODE) through NODZPP(NODE+1)-1 as a segment of NODZP(?), which in

turn lists the zones attached to each node. In this way we need no restriction on the connectivity of the system. Although all this information duplicates data contained in the arrays LNODE and ZNODE, it can be easily processed by node and thereby saves much searching time.

2.18 STANDARD INDEXES

We have tried, throughout SMART, to use consistent index names. Most modules declare a set of standard indexes, whether they are all used or not. Additional indexes are created as needed by appending to the standard index a one-digit suffix, such as LEG1 and LEG2 coming from LEG. The standard indexes are:

CARD.INPUT.FIELD (FLD)

LINK

NODE
FROM.NODE (NODE1)
TO.NODE (NODE2)

ZONE
FROM.ZONE (ZONE1)
TO.ZONE (ZONE2)

TIME.OF.DAY.INDEX (TIME)

DIRECTION (DIR)
- For FEEDER, DUMPER, and CBD, direction 1 is outbound and direction 2 is inbound.
- For LINKER modes, direction 1 is selected trip and direction 2 is reverse trip.
- For links, directions 1 and 2 are arbitrary.
- For output matrices, direction 3 is an accumulation area.

BACK.POINTER.INDEX (IPP)

TRIP.TYPE (TTYPE)

TRANSIT.MODE (MOD)
DEFINED.TRANSIT.MODE (DMOD)

CHARACTER.GROUP (CHAR)

MODE.SHARE.INDEX (MSI)

LINKER.TRANSIT.MODE.CATAGORY (MCAT)
- Fixed Route Bus = 1
- Light Rail = 2
- Heavy Rail = 3

TRANSIT.ROUTE.LEG (LEG)

The FORTRAN declaration of these standard indexes is:

```
C               DATA STRUCTURE STDIND AS OF 06/18/79
      INTEGER FLD, LINK, NODE, ZONE, NODE1, NODE2, ZONE1, ZONE2,
     1    TIME, DIR, IPP, TTYPE, MOD, DMOD, CHAR, MSI, MCAT, LEG
C                       DATA STRUCTURE STDIND END
```

3. PROGRAM STRUCTURE

Each SMART module is designed to operate as independently as possible in the performance of a specific task. Most of these modules can be grouped into one of nine major subsystems. The following sections present brief descriptions each of those subsystems. The symbols used in the program structure diagrams are explained in Exhibit 3.1

3.1 URBAN STRUCTURE SUBSYSTEM

Exhibit 3.2 shows the modules which maintain the urban structure. Keywords that define the urban network structure will activate the PROCESS.NETWORK (PRONET) module. This module controls network parameters such as ring radii, zone size, and link length. It examines all information on urban-related cards and prints error messages if mistakes are located. Depending on the modifications required, it may need to access such routines as UNDEFINE.NETWORK (UNFLAG), GENERATE.URBAN.STRUCTURE (GUSTRC), UPDATE.LINK (UPDLNK), UPDATE.ZONE (UPDZON), LINK.NUMBER (LINKNO), and ZONE.NUMBER (ZONENO) to help with the network modifications. Changes to radii, number of rings and corridors, and angles are made by PRONET itself, and no other modules are necessary. However, if the URBAN or NULLNET keywords are encountered, PRONET must call UNDEFINE.NETWORK (UNFLAG) to clear all previous urban parameters and to issue warning messages.

For zone or link updating, SMART requires that the updates be made to an existing structure. Therefore, PRONET checks the HAVE.URBAN.STRUCTURE.ALREADY? (HUSTRC) indicator, which is a member of the URBAN.STRUCTURE (USTRUC) data structure. If HUSTRC is .FALSE., no network exists and PRONET must call GENERATE.URBAN.STRUCTURE (GUSTRC) to initialize the network. If a NULLNET card is in effect, then it is initialized to an empty network, with zero zones, nodes, and links. If a NULLNET card is not active, it is initialized to a ring/corridor-type system using the current values for the number of rings and corridors and ring radii and corridor angles.

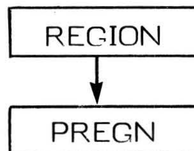
Each zone or link has a unique index. For a ring/corridor structure, this index can be computed from the ring number, corridor number, number of rings and zones, and the left/right or radial/circumferential location description. This happens in the modules ZONE.NUMBER (ZONENO) and LINK.NUMBER (LINKNO), and in NODE.NUMBER (NODENO) for nodes, when a ring/corridor system is created or when LINKSET or ZONESET cards are encountered. When a zone or link identifier is given, on the ONEZONE or ONELINK cards, LIST.LOOKUP (LIST8) must be used to find the index in the list of

EXHIBIT 3.1

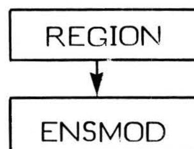
PROGRAM STRUCTURE SYMBOLS

A single-walled box represents a module; for example, REGION appears as:

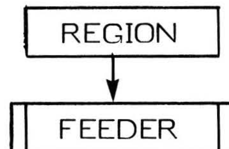
Module activations, either through a CALL statement or a function reference, appear as arrows from the activator to the activated. For example, REGION calls PRINT.REGION.PARAMETERS (PREGN):



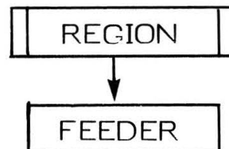
If there are other activations of this module not shown on this diagram the activated module appear with a circle on its corner. For example, REGION is one of many modules which calls ENSURE.MODE (ENSMOD):



When the activated or activator is part of some other subsystem, then the subsystem name appears in a double-walled bax. For example, REGION calls FEEDER. In the REGION diagram, there appears:



In the Feeder diagram, there appears:

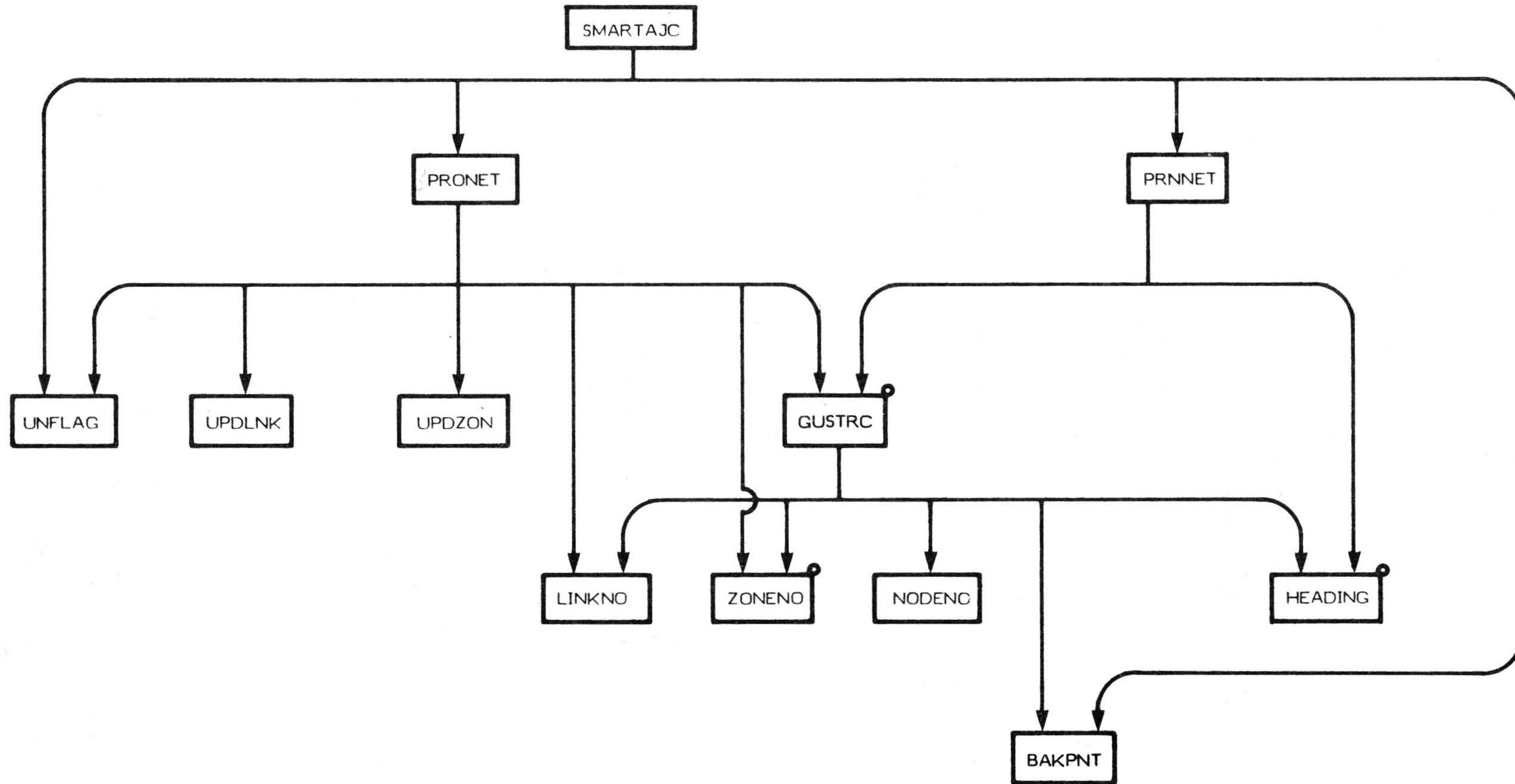


An activation of a separate subsystem hown in a diagram never shows all the possible ways that subsystem can be activated. Hence all double-walled boxes have an implied attached circle.

The module SMART.MAIN.PROGRAM (SMARTAJC) appears in all subsystem diagrams except the one for results printing, which is activated only through one of the other subsystems.

EXHIBIT 3.2

URBAN STRUCTURE SUBSYSTEM



identifiers. Once the link or zone is known, UPDLNK and UPDZON will be called to make the required modifications. Zone and link characteristics are modified only by the UPDATE.LINK (UPDLNK) and UPDATE.ZONE (UPDZON) routines which are called only by PRONET. Options and parameters are examined, and zone and link characteristics are set accordingly.

No modifications to the network can be made until GENERATE.URBAN.STRUCTURE (GUSTRC) has been invoked and HAVE.URBAN.STRUCTURE (HUSTRC) set to .TRUE. Once this is done, and until the next URBAN or NULLNET card is encountered, SMART will avoid any network regeneration, which would erase the network modifications made already.

Link, zone and node pointers are set up by subroutine BACK.POINTER (BAKPNT). These pointers are necessary for finding routes through the network.

Note tht GUSTRC is called, not only by PRONET, but by other routines as well. In particular, it is accessed by any module which requires that an urban structure be present before computations can be performed.

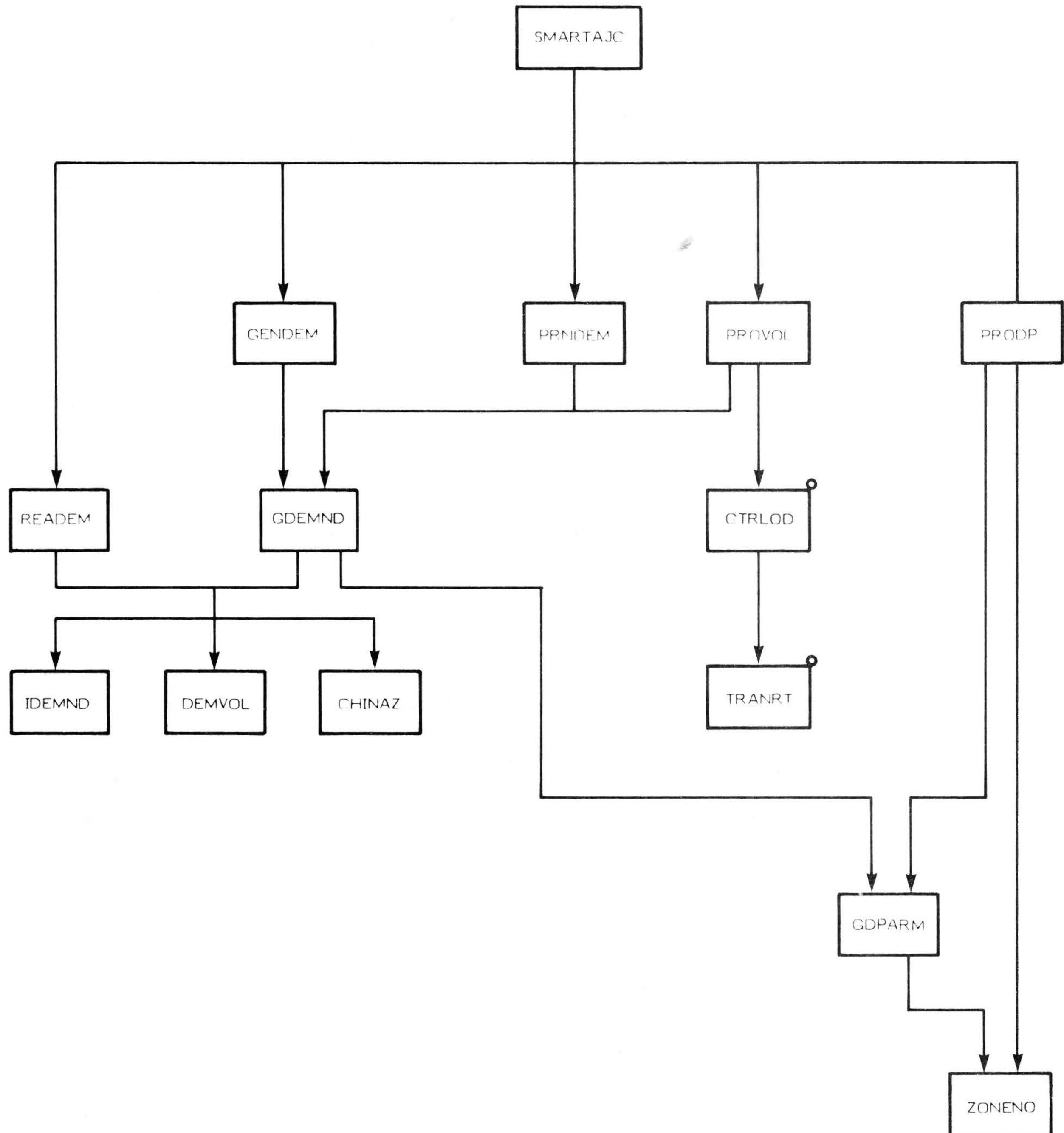
3.2 DEMAND SUBSYSTEM

This subsystem, shown in Exhibit 3.3, can be roughly separated into three segments: one dealing with demand parameters, one dealing with demand generation, and one dealing with demand volume updating. Default demand parameters are contained in the matrix DEFAULT.ZONAL.DEMAND.PARAMETERS (DZDPARM) and the vector EXTRA.DEMAND.PARAMETERS (DXPARM), assigned to zones by module GENERATE.DEMAND.PARAMETERS (GDPARM) and updated by subroutine PROCESS.DEMAND.PARAMETERS (PRODP). All demands and trip volumes are either read in from an external file or calculated from zonal demand parameters, which contain information on zonal population and employment density, trip generation rates, and trip lengths. The default demand generation procedure is subroutine GENERATE.DEMAND (GDEMND), which computes trip rates from the zone's population, employment, and demand parameters, and distributes them to destination zones based on an exponentially decreasing function of distance. External demand data are read by subroutine READ.DEMAND (READEM). All trip volume updating is done by module PROCESS.VOLUME.CHANGES (PROVOL).

Before demand parameters can be changed, two tasks must be completed: An urban structure must be available, and default demand parameters must have been assigned. Two variables indicate whether these tasks have been attended to: HAVE.URBAN.STRUCTURE.ALREADY? (HUSTRC) and HAVE.DEMAND.PARAMETERS? (HDPARM). Both are logical variables, the former in the data structure URBAN.STRUCTURE (USTRUC) and the latter in DEMAND.PARAMETERS (DPARM). If HUSTRC is .TRUE., an urban structure is

EXHIBIT 3.3

DEMAND SUBSYSTEM



available and complete; if it is .FALSE., SMART has not yet created the urban network, and GUSTRC must be called. Similarly, if HDPARM is .TRUE., demand parameters have been assigned; otherwise, they have not been loaded yet, and GDPARM must be called.

For the ring/corridor structure, demand parameters are assigned on the basis of ring/corridor location; that is, whether the zone is in Ring 1, 2, 3 and so forth, while arbitrary network demand parameters are all defaulted to Ring 3 default values. ZONE.NUMBER (ZONENO) is used by GDPARM to determine the zone index of each ring/corridor zone.

Demand volumes are obtained either through the default generation procedure in GENERATE.DEMANDS (GDEMND), from an external file through subroutine READ.DEMAND (READEM), or through a special demand generation procedure installed by a programmer (see the chapter 'PROGRAM ALTERATIONS'). In any case, routes must have already been generated because they relate node-to-node traffic to link traffic. For demand generation procedures, demand parameters must also have been loaded. Therefore, if they are unavailable, GENERATE.ROUTES (GROUTS) or GENERATE.DEMAND.PARAMETERS (GDPARM) must be called to perform the required computations. Any demand-producing module will call INITIALIZE.DEMAND (IDEMND) to set all demands initially to zero, then compute (for generation) or read in (for READEM) the trip distribution and aggregate it using the DEMAND.VOLUME (DEMVOL) procedure.

Modifications to trip volume are made by subroutine PROCESS.VOLUME.CHANGES (PROVOL), which is called only by SMART.MAIN.PROGRAM (SMARTAJC). This module first ensures that an urban structure and demand volumes already exist (if not, it calls GUSTRC and GDEMND). Then it introduces the changes requested by the input cards. For transit link traffic changes through the TRAFFIC card, it may have to call the GENERATE.TRANSIT.LOAD (GTRLOD) routine to compute initial transit loads before making modifications.

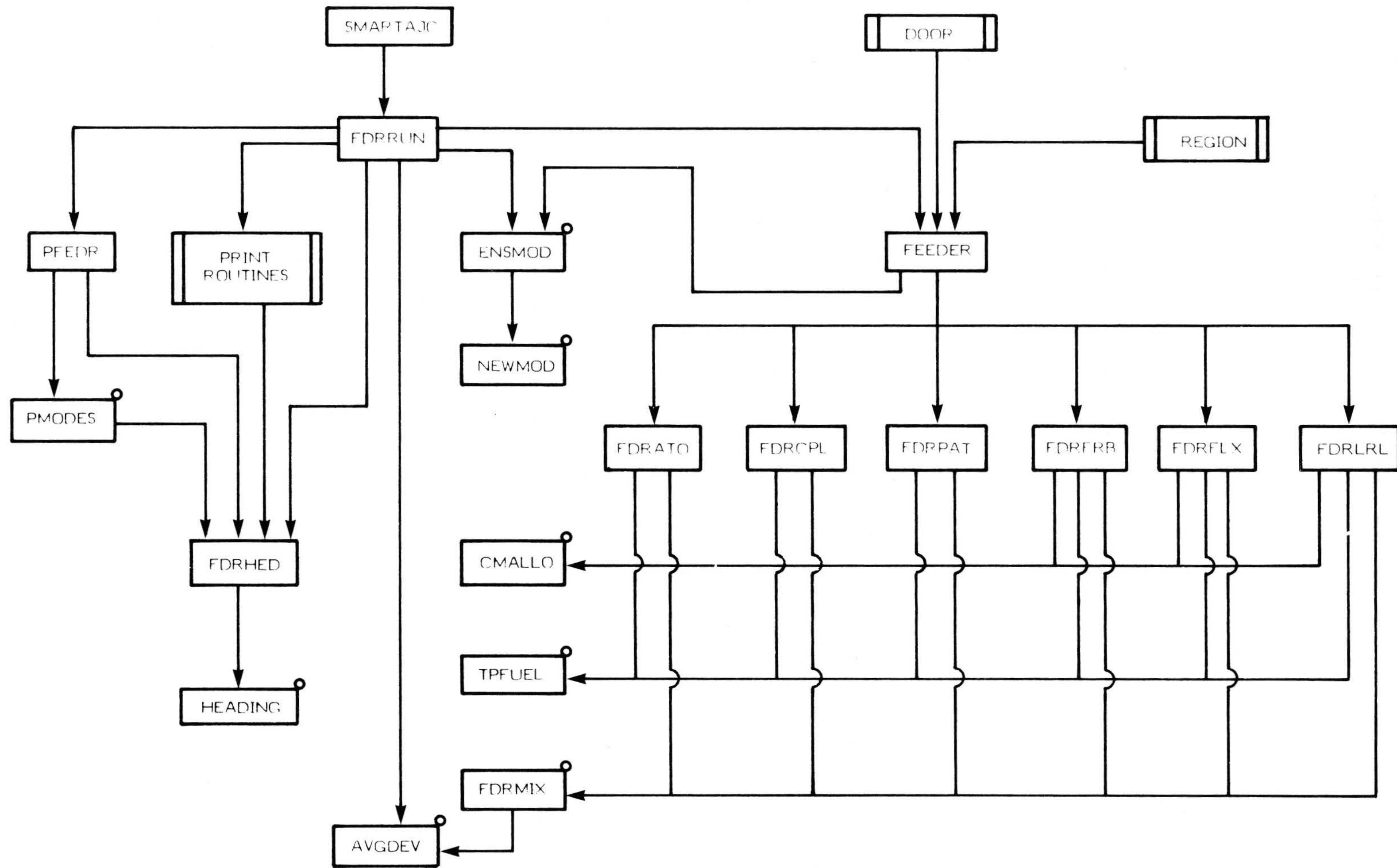
3.3 FEEDER SUBSYSTEM

The FEEDER subsystem is portrayed in Exhibit 3.4 The driving routine is the FEEDER.RUN (FDRRUN) module, which is invoked only by SMART.MAIN.PROGRAM (SMARTAJC) upon encountering the FEEDER keyword. The FDRRUN module checks all input information, loads default modes if the automatic mode selection process is in effect, and suppresses analysis if errors have been found. FDRRUN calls FEEDER to perform the actual zonal analysis, then sums the output from the FEEDER routine to obtain bidirectional totals. Printing routines are then called to display the results of the model.

The first task of FEEDER is to call ENSURE.MODES.SELECTIONS (ENSMOD) to ensure that at least one auto, one carpool, and one transit mode have been selected for analysis. If a CORRIDOR analysis is required, FEEDER ensures light-rail selection as well. Modal parameters for any newly selected mode are initialized in

EXHIBIT 3.4

FEEDER SUBSYSTEM



subroutine NEW.MODE (NEWMOD). FEEDER then calls the individual transit mode routines to compute performance measures.

Individual mode routines are FEEDER.AUTO (FDRATO), FEEDER.CARPOOL (FDRCPL), FEEDER.PRIVATE.AUTO (FDRPAT), FEEDER.FIXED.ROUTE.BUS (FDRFRB), FEEDER.FLEXIBLE.ROUTE (FDRFLX), and FEEDER.LIGHT.RAIL (FDRLRL). These contain the mathematical relationships that form the basis for FEEDER system analysis. They access several modules that help in computing performance measures: CAPITAL.MAINTENANCE.ALLOCATION (CMALLO), TRIP.FUEL (TPFUEL), FEEDER.MIX.DENSITY.RESULTS (FDRMIX), which properly sum and average results from FEEDER dual density analyses, and AVERAGE.STANDARD.DEVIATIONS (AVGDEV), which computes the standard deviation of travel time for dual density analyses.

FEEDER.HEADINGS (FDRHED) and HEADINGS (HEADNG) are called by print routines to display heading labels.

3.4 DUMPER SUBSYSTEM

The DUMPER subsystem, shown in Exhibit 3.5, is structured in much the same way as the FEEDER block. It is somewhat less complex, since there are fewer modes to model, and it does not have to contend with dual density analyses. The individual mode routines consist of DUMPER.AUTO (DPRATO), DUMPER.CARPOOL (DPRCPL), and DUMPER.FIXED.ROUTE.BUS (DPRFRB). These contain the mathematical relationships that form the basis for modeling and evaluating DUMPER service. They access the modules TRIP.FUEL (TPFUEL), which computes fuel consumption measures, and CAPITAL.MAINTENANCE.ALLOCATION (CMALLO), which allocates capital and maintenance costs across the different time periods in a day.

3.5 CBD SUBSYSTEM

The CBD block, which also resembles the FEEDER subsystem, is shown in Exhibit 3.6 There are four individual mode routines: CBD.AUTO (CBDATO), CBD.CARPOOL (CBDCPL), CBD.FIXED.ROUTE.BUS (CBDFRB), and CBD.AUTOMATED.GUIDEWAY.TRANSIT (CBDAGT), which contain the mathematical relationships that describe system operations. Two modules found here have no corresponding FEEDER version: CONGESTED.ZONE.SPEED (CZSPED) and AGT.EFFECTIVE.VELOCITY (AGTVEL). CZSPED calculates the congested vehicle speed based on CBD traffic volume and Smeed's congestion relation.[1] AGTVEL, accessed only by CBDAGT, computes AGT vehicle speed based on station spacing. TRIP.FUEL (TPFUEL) and CAPITAL.MAINTENANCE.ALLOCATION (CMALLO) are used for computing fuel

---

[1] Smeed, R.J., "Traffic Studies and Urban Congestion," Journal of Transport Economics and Policy, January 1968.

EXHIBIT 3.5
DUMPER SUBSYSTEM

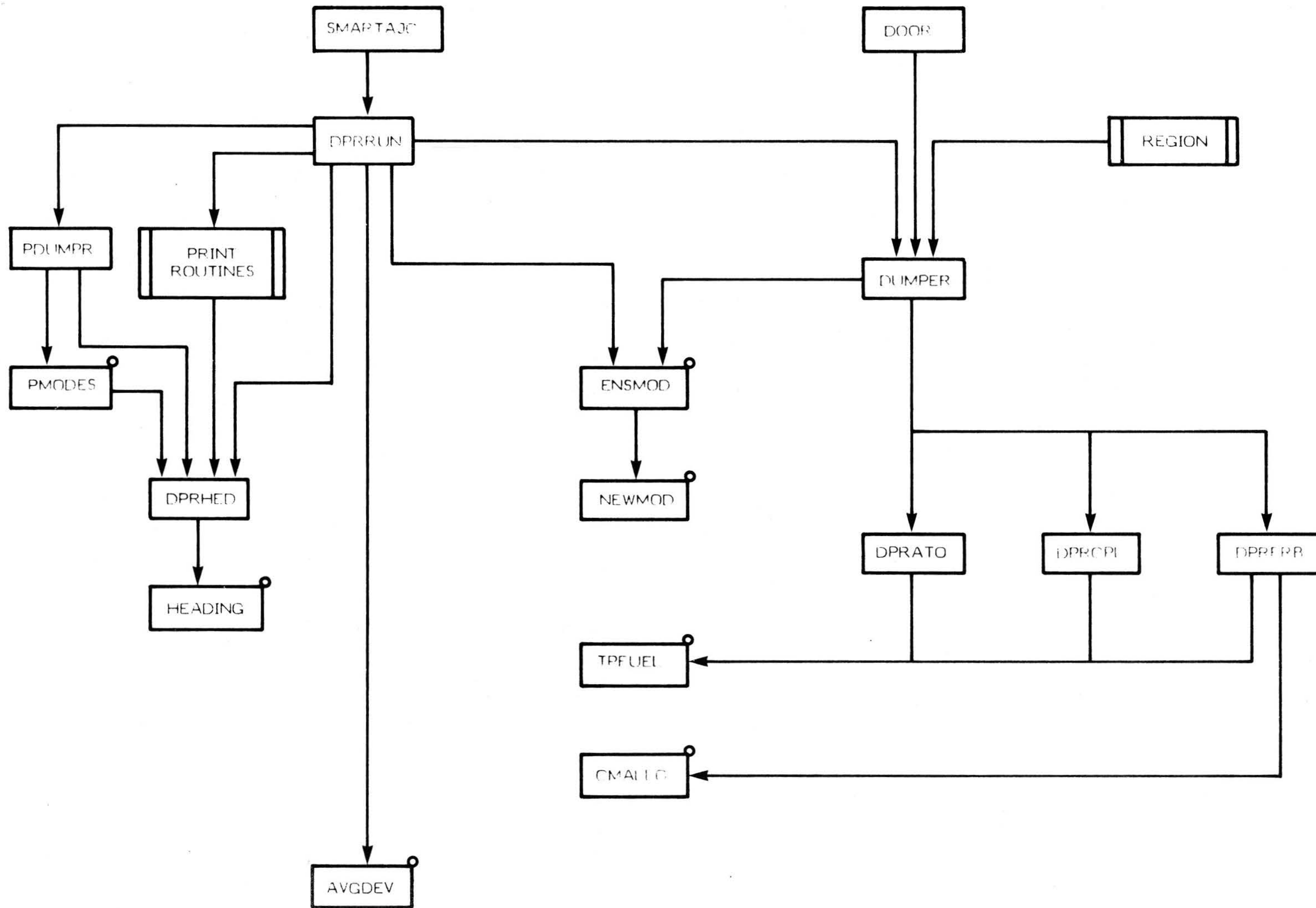
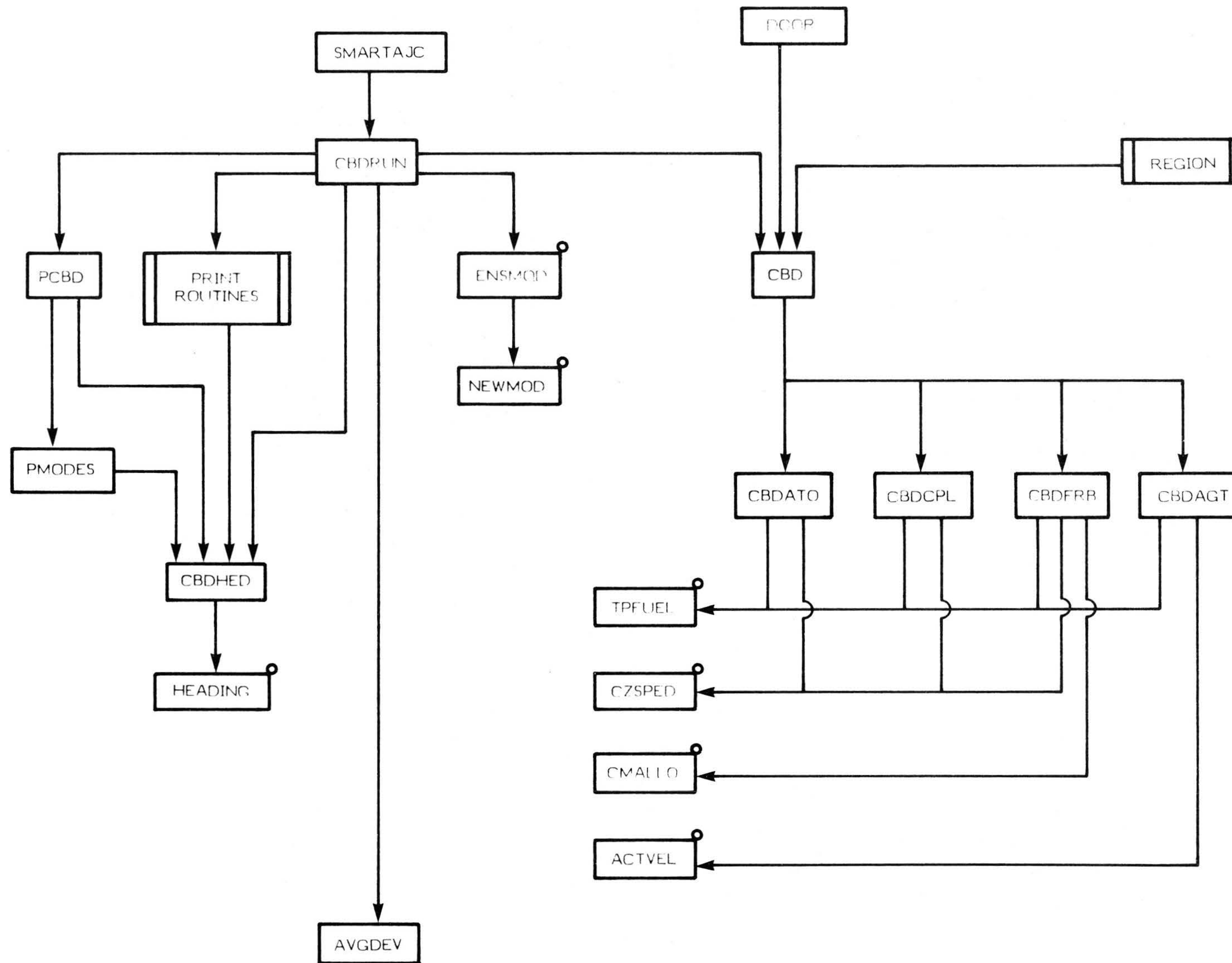


EXHIBIT 3.6
CBD SUBSYSTEM



consumption and cost allocation.

3.6 LINKER SUBSYSTEM

This subsystem is diagrammed in Exhibit 3.7 Upon encountering the LINKER keyword, SMART.MAIN.PROGRAM (SMARTAJC) activates the LINKER.RUN (LKRRUN) routine, which coordinates SMART's LINKER analysis procedures. This module checks the input data and suppresses analysis if errors are found. It also checks transit loads and, if they are undefined, GENERATE.TRANSIT.LOAD (GTRLOD) will be called to compute transit pick-ups, drop-offs and loading for all network links. Next, TRANSIT.ROUTE (TRANRT) is called to compute the transit route between the origin and destination nodes specified on the keyword card. LINKER is then called to perform line-haul analysis. Results from this routine are summed and printed by LINKER.RUN (LKRRUN) through the printing routines.

LINKER's first task is to ensure that an urban structure and a set of demand volumes have been generated. If not, it calls the GENERATE.URBAN.STRUCTURE (GUSTRC) and GENERATE.DEMAND (GDEMND) procedures to compute the required inputs. ENSURE.MODES.SELECTIONS (ENSMOD) is called to make certain that at least one auto and one carpool mode have been selected. All newly selected modes will have their parameters initialized by subroutine NEW.MODE (NEWMOD). LINKER then examines each leg on the shortest interconnecting path and ensures that its mode category has been selected. Line-haul performance measures for automobiles and carpools are obtained by calling subroutine LINKER.CAR (LKRCAR). Apart from the differing demand rates and vehicle speeds (carpools can use preferential lanes), the LKRCAR subroutine handles both modes in the same manner. Next, LINKER computes transit demand and determines if it is large enough to justify express bus service between the designated nodes. It calls TRANSIT.NEEDS (TRANED) to compute headways, loads, speeds, times and other transit performance measures. Outputs are then allocated by mode categories, and an appropriate portion is billed to trips made between the designated origin and destination modes. Congested link speeds are computed by subroutine CONGESTED.LINK.SPEED (CLSPED), which is called by both TRANED and LKRCAR.

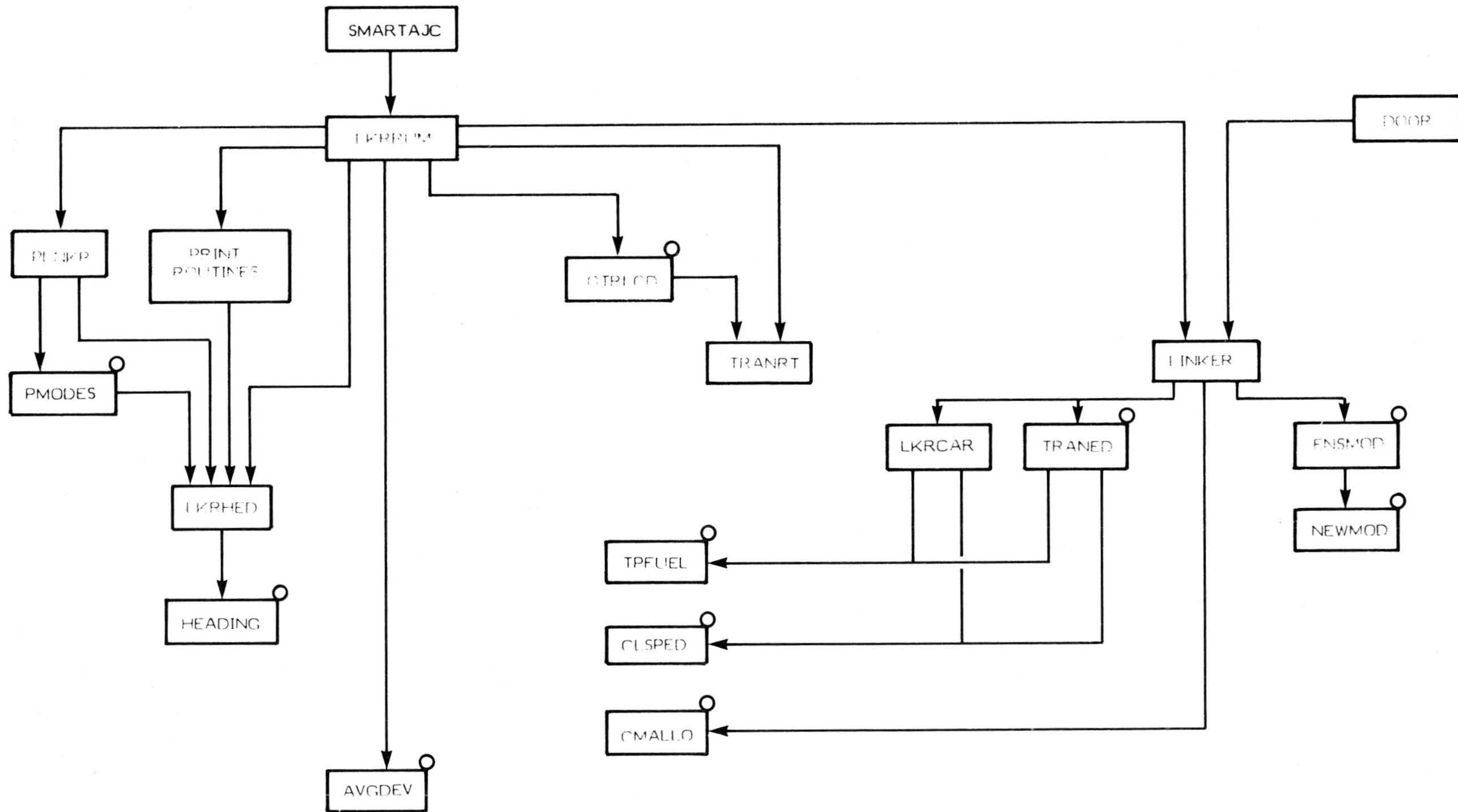
Subroutines LINKER.HEADINGS (LKRHED) and HEADINGS (HEADNG) are called by the printing routines to display heading information.

3.7 DOOR.TO.DOOR SUBSYSTEM

These modules are shown in Exhibit 3.8 Once the DOOR keyword is encountered, the DOOR.TO.DOOR subsystem uses subroutine DOOR.TO.DOOR.RUN (DORRUN) to ensure that urban structure and demand volumes have been generated. If not, DORRUN calls the GENERATE.URBAN.STRUCTURE (GUSTRC) and GENERATE.DEMAND (GDEMND)

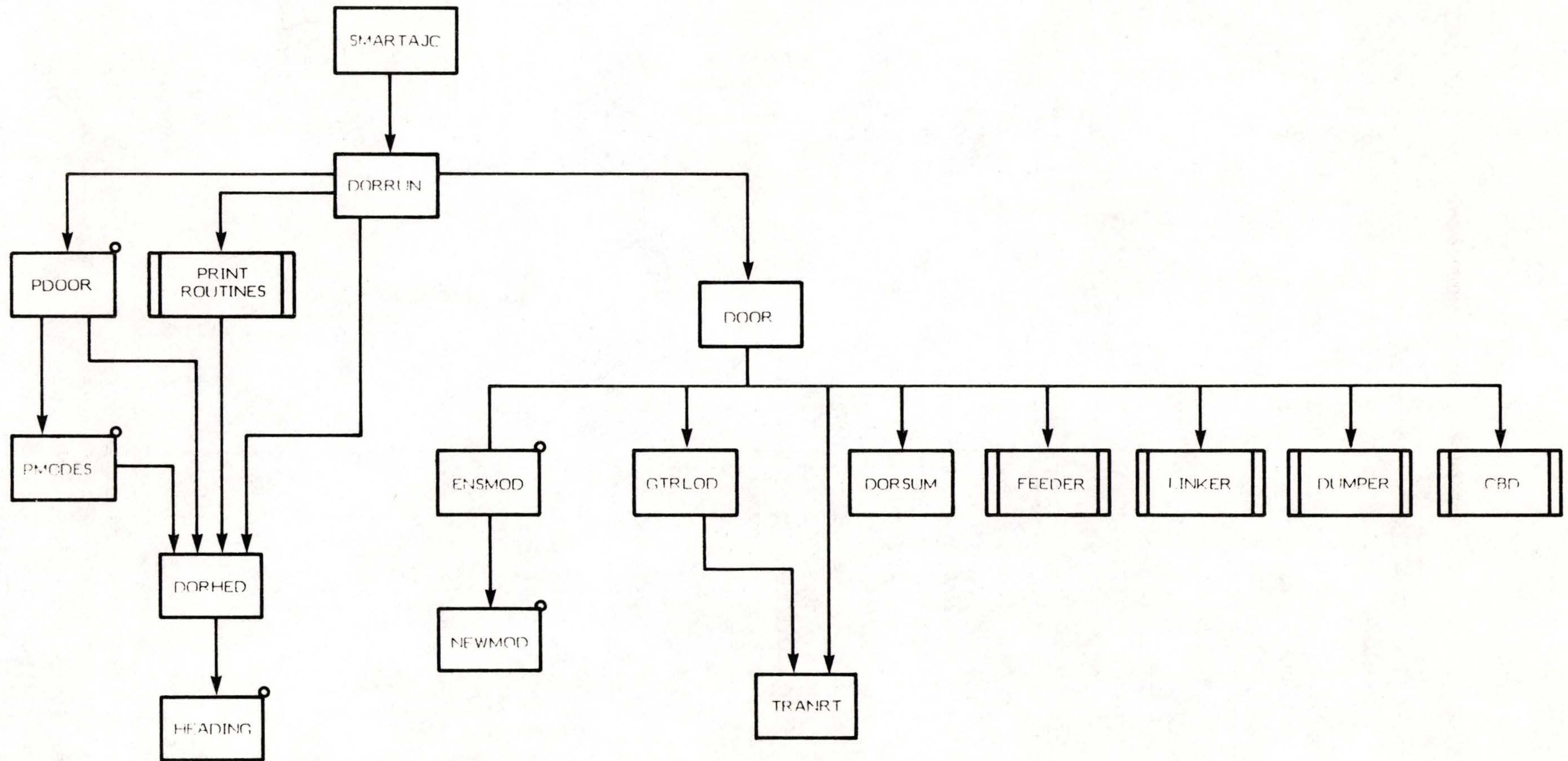
EXHIBIT 3.7

LINKER SUBSYSTEM



EXHIBTI 3.8

DOOR.TO.DOOR SUBSYSTEM



procedures to create the required data. All inputs on the DOOR card are checked, and analysis is suppressed if errors are found. DORRUN calls subroutine DOOR.TO.DOOR (DOOR) to compute door-to-door performance measures. These are displayed by DORRUN through the printing routines.

Module DOOR calls ENSURE.MODES.SELECTIONS (ENSMOD) to ensure that at least one auto, one carpool, and one transit mode are selected for analysis. All newly selected modes have their parameters initialized in subroutine NEW.MODE (NEWMOD). If LINKER transit loads have not yet been generated, DOOR will call GENERATE.TRANSIT.LOAD (GTRLOD) to compute vehicle loading. TRANSIT.ROUTE (TRANRT) will also be called to generate the door-to-door line-haul route. DOOR creates four "dummy" modes for storage allocation: AUTO.TOTAL (ATOTOT), CARPOOL.TOTAL (CPLTOT), TRANSIT.TOTAL (TRNTOT), and MODE.TOTAL (MODTOT). The first three accummulate results from origin to destination for the corresponding modes; the last creates a temporary storage location to record separately the performance measures of the origin and destination zone transit service. In the event that the origin and the destination have the same mode (for example, both have FEEDER fixed-route bus), this extra location assures that the performance measures of both modes will be saved (by time, direction and the mode share index) and available for later use. The FEEDER and DUMPER or CBD subsystems are called to calculate zonal measures, depending on the type of origin and destination zones. Results from the zonal analysis are added up in subroutine DOOR.TO.DOOR.SUM (DORSUM). LINKER is then called to compute line-haul measures, and these are added to the zonal outputs to provide a door-to-door picture of urban travel.

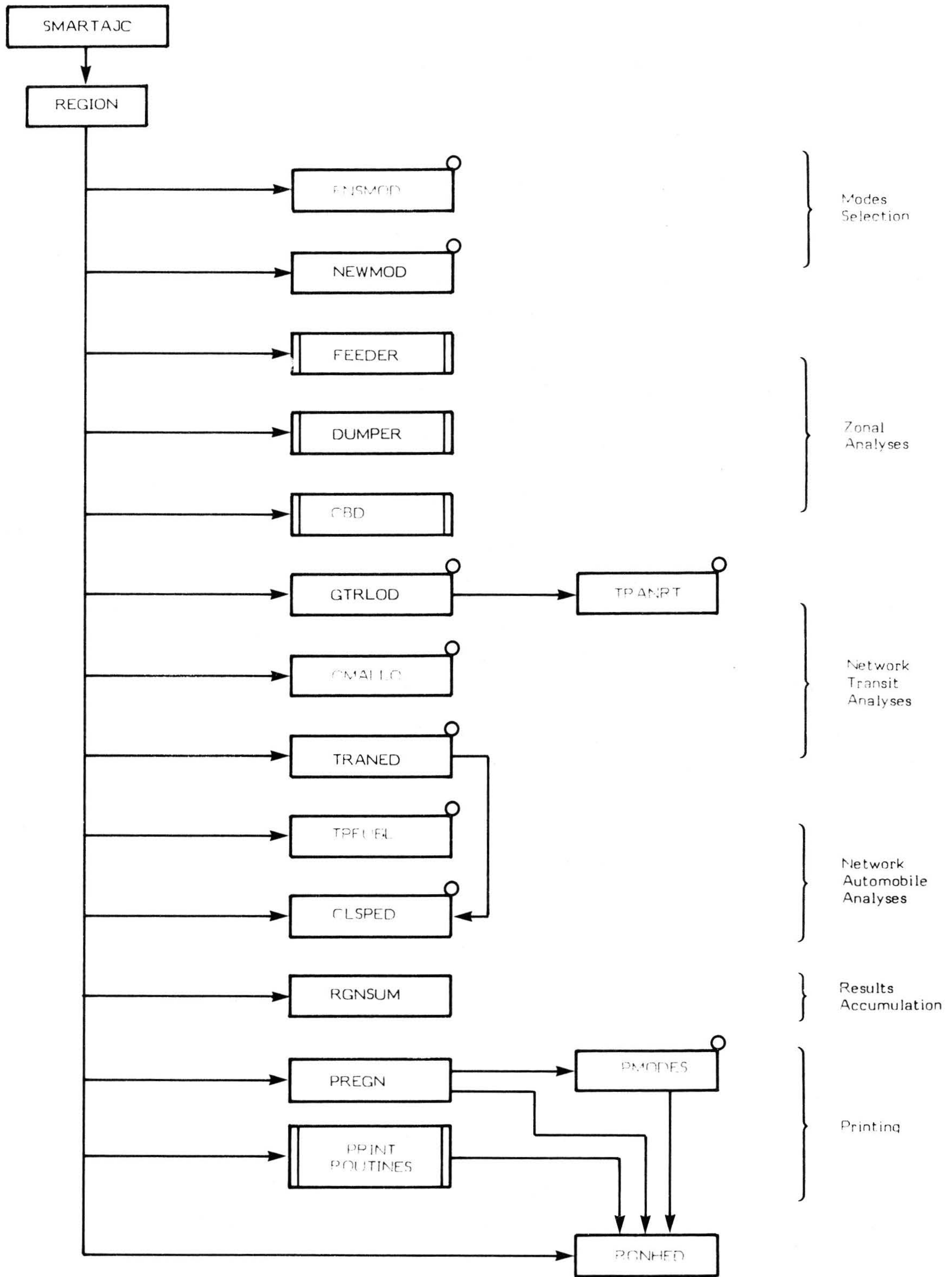
One of the resulting performance measures is passenger pick-ups per minute. For DOOR, this should be the zone-to-zone demand rate. However, to minimize storage requirements, this statistic is not saved by the demand generation procedures of SMART, and hence cannot be printed. Instead, subroutine PRINT.DOOR.TO.DOOR.PICKUPS (PDORPK) displays the total number of outbound and inbound trips (pick-ups) in the origin/destination zones and pick-ups on the interconnecting links. These flows are, of course, larger than the volume of zone-to-zone trips, since they include trips between other origins and destinations as well.

3.8 REGION SUBSYSTEM

The REGION modules are shown in Exhibit 3.9 The tasks in this subsystem are coordinated or performed by subroutine REGION. To perform a regionwide analysis, an urban structure and demand volumes must first have been specified. If they are not, REGION will call the GENERATE.URBAN.STRUCTURE (GUSTRC) and GENERATE.DEMAND (GDEMND) procedures to create the required data. For each zone, REGION ensures a designated transit mode. If no transit mode has been designated, REGION will select fixed-route bus. For each basic analysis type, exactly one auto and one carpool mode will be selected. REGION also creates three "dummy" modes: AUTO.TOTAL (ATOTOT), CARPOOL.TOTAL (CPLTOT), and

EXHIBIT 3.9

REGION SUBSYSTEM



TRANSIT.TOTAL (TRNTOT) to provide space for summing modal outputs. ATOTOT contains total regionwide performance measures for auto modes; CPLTOT for carpool mode; and TRNTOT for transit modes. Although the modal parameters for these dummy modes are unused, parameter values are set in subroutine NEW.MODE (NEWMOD) for these dummy modes since some FORTRAN monitors object to uninitialized variables.

Depending on the zone type, either FEEDER and DUMPER or CBD will be called to perform the actual zonal analysis. Once zonal outputs have been generated, REGION.MODAL.SUM (RGNSUM) is called to accummulate results of the zonal analysis.

If link loads are absent, REGION will call GENERATE.TRANSIT.LOAD (GTRLOD) to compute the required loading. Since loading depends on the transit routes selected between nodes, GTRLOD must access subroutine TRANSIT.ROUTE to get routes. For each link and for each loaded LINKER transit mode, REGION calls module TRANSIT.NEEDS (TRANED) to compute performance measures for that link. Temporal allocation of costs is determined by module CAPITAL.MAINTENANCE.ALLOCATION (CMALLO), while the function TRIP.FUEL (TPFUEL) and CONGESTED.LINK.SPEED (CLSPED) calculate fuel consumption and congested link speed, respectively. For the LINKER auto and carpool modes, performance measures are calculated directly in REGION.

Finally, all regionwide results are printed.

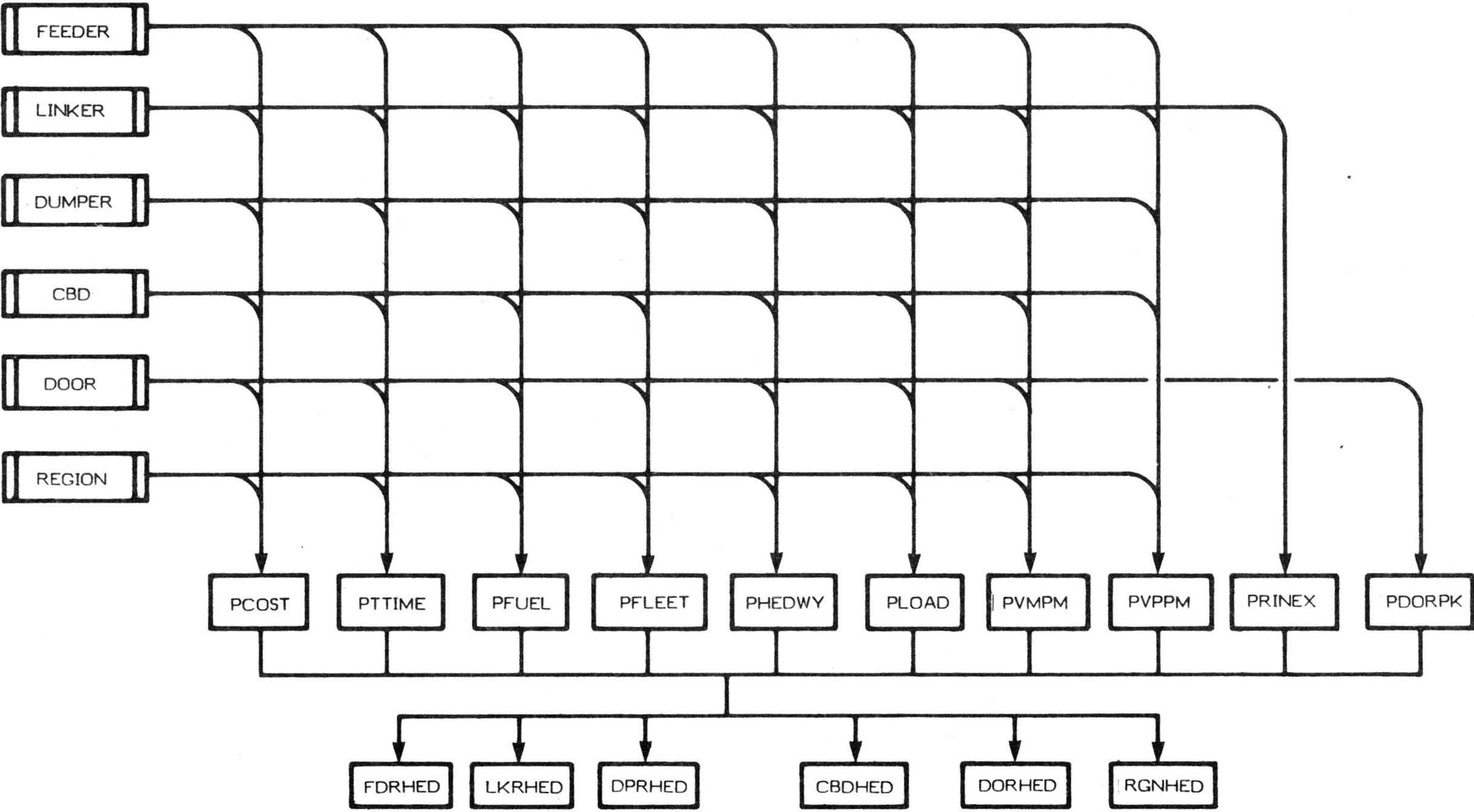
3.9 RESULTS PRINTING SUBSYSTEM

The modules which print SMART's findings are shown in Exhibit 3.10, and their interconnections indicated. This subsystem consists primarily of PRINT.COST (PCOST), PRINT.TRAVEL.TIME (PTTIME), PRINT.FUEL.CONSUMPTION (PFUEL), PRINT.FLEET.SIZE (PFLEET), PRINT.HEADWAY (PHEDWY), PRINT.VEHICLE.LOADING (PLOAD), PRINT.VEHICLE.MILES (PVMPM), and PRINT.PICKUPS (PVPPM). In general, all of these routines are used for all of the results printing by all of the analysis routines (FEEDER, DUMPER, CBD, LINKER, DOOR, and REGION). The exceptions are: PRINT.PICKUPS (PVPPM) is not used by DOOR.TO.DOOR, which calls PRINT.DOOR.TO.DOOR.PICKUPS (PDORPK) instead. Also, LINKER calls PRINT.EXPRESS.STATUS (PRINEX) to show whether line-haul fixed-route buses are multi-stop or express.

The heading routines LINKER.HEADINGS (LKRHED), FEEDER.HEADINGS (FDRHED), DUMPER.HEADINGS (DPRHED), CBD.HEADINGS (CBDHED), REGIONWIDE.HEADINGS (RGNHED), and DOOR.TO.DOOR.HEADINGS (DORHED) are called by the printing routines, depending upon which analysis block invoked the print. The method used is to pass the name of the printing subroutine as a parameter to each of the results printing modules. The overall heading for the SMART model is printed by subroutine PAGE.HEADINGS (HEADNG).

EXHIBIT 3.10

RESULTS PRINTING SUBSYSTEM



Note that the LINKER subsystem does not call PLOAD, and that the DOOR subsystem does not call PVPPM.

Actual calls to the heading routines follow the same pattern as the calls to the print routines (e.g., CBDHED is called if and only if the print routine was activated by the CBD subsystem.)

4. MACHINE INDEPENDENCE AND INSTALLATION

SMART code standards were selected for:

- Portability;
- Reliability; and
- Low execution cost.

A truly portable program is one that is machine-independent and that can be transferred from one computer or input/output device to its functional equivalent without any reprogramming or variation in results. This ideal state can never be completely realized in a program of the size and complexity of SMART, even though every effort has been made to increase its portability. The basis for the FORTRAN used in SMART is American National Standards Institute (ANSI) report X3.9-1966, commonly known as FORTRAN-66. Certain extensions to that standard were used to increase program reliability, where these extensions are believed to be widely available. Also, some functions can only be performed reasonable efficiently by machine-dependent code, but equivalent code exists for virtually all computers.

4.1 EXTENSIONS TO FORTRAN-66

The following extensions to FORTRAN-66 are used in SMART:

1. Hollerith literals may be enclosed in apostrophes (single quotes).

2. An array may be initialized in a DATA statement by merely giving its name, with the number of values equal to the number of elements in the array.

3. Double-precision variables are assumed to hold at least eight characters. No assumption is made about the total number of characters, the placement of these characters, or the padding. Character-containing variables are only compared for equal or not equal.

4. FORTRAN-66 requires that both subprograms and COMMON blocks be overlayable. All SMART subprograms are overlayable (that is, none remember data from one call to the next). However, no COMMON block is overlayable; each must remain in core during the entire program execution. On any machine, there should be a way to ensure this, at least by adding otherwise unnecessary declarations to the main program.

5. The maximum number of dimensions for any array in SMART is 4, whereas FORTRAN-66 permits no more than 3. Affected arrays are those that contain performance and output measures from the model, such as cost, fuel consumption, vehicle loading, and so forth. These are dimensioned by time of day, direction, mode share index, and mode.

4.2 SPECIFIC VIOLATIONS OF FORTRAN-66

1. Every module begins with the statement:
```
     IMPLICIT LOGICAL * 1 (A-Z) .
```
This permits compile-time detection of most spelling errors. The default is never deliberately used, and these statements may be removed if necessary.

2. The main program contains an END= option on the read statement, as does the module READDEM. SMART specifications require end-of-file indicator records, so these END= uses function only to allow SMART, rather than the operating system, to detect the error. The END= options may be removed, with no affect on the processing of correct input decks.

3. The module MKREAL is machine-dependent. This subroutine examines an eight character field, stored in a DOUBLE PRECISION variable, and returns the equivalent real value. It is, in effect, a core-to-core read with format F8.0. The error indicator that is returned if the field is not numeric is only used for detecting user errors. Thus MKREAL would be adequate, if not pleasant to use, if it simply aborted when an error was detected. In an extreme case, MKREAL could be implemented using the following very slow code:

```
      WRITE (unit, 210) DWORD
210   FORMAT (A8)
      REWIND unit
      READ (unit,220) VALUE
220   FORMAT (F8.0)
      ERROR=.FALSE.
```

4. LIST.LOOKUP (LIST8) is moderately machine-dependent. This dependency arises because the list is allowed to contain zero entries, and the list is not always the same length. One FORTRAN monitor may require it to be dimensioned to NELE (the number of elements in the list) even though NELE is sometimes zero. Other monitors may insist that it be dimensioned to the largest possible length, such as 9999, even though the array is actually smaller than that. Some monitors will not accept either declaration, but will tolerate a declaration of (1) as an indication of a dynamic dimension. Some experimentation may be needed to see what your run-time monitor can live with.

5. GETDAT returns the current date in eight characters, of the form "MM/DD/YY." This is very system-dependent. The field is used only for printing, so GETDAT may return eight blanks, in which case the printout will be undated.

6. In the archived network/demands system, speed is achieved by equating real vectors to the entire data structures and using unformatted I/O for that vector. The size of the vector assumes that DOUBLE PRECISION variables are twice as large as REAL variables, and that INTEGER and LOGICAL variables are the same size as REAL variables. If this is not the case, the length of arrays GDEM in DEMAND and GNET in URBAN.STRUCTURE (USTRUC) must be changed.

4.3 PROGRAM SIZE

SMART is a rather large program. The data structures occupy 38,000 words, and the executeable code approximately 60,000 words. On an IBM/370-type system, using the FORTRAN H compiler with full optimization, total module size, including data, program code, and FORTRAN monitor, was 361,112 bytes (90,278 words). When the WATFIV debugging compiler was used, total memory required shot up to over 720,000 bytes.

In most runs, not all blocks are executed. Hence, memory requirements can be reduced via overlay techniques. In such a case, the following guidelines are useful:

1. COMMON blocks must be part of the root segment.

2. Overlay segment boundaries can generally follow analysis block boundaries.

3. A regionwide analysis could become extremely expensive, since it repeatedly calls on most of the other blocks.

4.4 ACCURACY

SMART was developed on an IBM/370-type system. On these computers, single-precision floating point numbers have 21 to 24 bits (6 to 7 decimal digits) of accuracy, depending on the value. This is about the minimum accuracy found on any computers, and SMART had no trouble with truncations or round-off errors. Thus the floating point word size should not be a consideration on any machine.

However, IBM/370-type systems allocate 31 bits for the value of an integer, plus one sign bit. Some systems default to only 15 bits for integer magnitudes. Although no problems are forseen in such a case,

such a conversion has not been tested and the installer is advised to proceed with caution.

5. PROGRAM ALTERATIONS

Great care has been exercised in the design of the SMART model to give it sufficient breadth and depth to handle a large variety of urban transport problems without any changes to the code. Nonetheless, situations may arise that require alterations to the SMART program in order to obtain a more satisfactory representation of the urban travel problem under study. For example, one case arose wherein traffic patterns on highways were so unusual that SMART's standard congestion/speed relation was unable to reflect the situation adequately. One module had to be replaced to handle the problem.

SMART's design is highly modular, with each module's task precisely defined and isolated. Thus, many changes to the program can be easily made because they are localized and affect only one or two modules. Other changes are spread throughout the program and, because of complex interdependencies, are risky as well as difficult to implement. Any change, however minor, will involve a certain amount of risk and, hence, should be avoided unless it improves program performance substantially.

One can make a variety of changes to SMART. This chapter discusses the changes that are most likely to be needed, and which can be accomplished without major expense.

5.1 DEFAULTS

Perhaps the easiest changes to SMART constitute alterations to the default values of parameters. However, such changes are rarely needed, since the user can override default values whenever needed. The values in the data structure FIXED cannot be reset by the user, and so it may at times be necessary to change these fixed parameters. Cost parameters are the most likely candidates for change. If inflation continues, these parameter defaults may need regular updating.

5.2 SIZE LIMITS

SMART limits the number of nodes, zones, links and other entities that it will handle. Initial limits have been set by SYSTAN to achieve the flexibility needed for realistic representation of a variety of urban settings. In its present form, SMART's data structures occupy 38,000 words of storage, and the object code requires roughly 60,000 additional words. With this large space usage, one hesitates to make any changes that would increase memory requirements.

Many of the data structure arrays, such as the number of selected links and zones, are dimensioned to certain maxima. Some of these maxima can easily be modified, while others are less accessible. Those that are not very accessible, such as the number of time periods or the number of LINKER mode categories, are so embedded in the code that only detailed alteration of the program can change them. Attempts to modify these parameters are discouraged, and we will not discuss such complex changes in this manual. Limits that can easily be modified include MAXIMUM.NUMBER.OF.ZONES (MXZONE), MAXIMUM.NUMBER.OF.LINKS (MXLINK), MAXIMUM.NUMBER.OF.NODES (MXNODE), MAXIMUM.NUMBER.OF.CORRIDORS (MXCOR), MAXIMUM.NUMBER.OF.RINGS (MXRING), and MAXIMUM.NUMBER.OF.MODES (MXMOD).

The process of increasing a SMART limit can be outlined as follows:

1. Identify all related limits which must be raised as well. For example, if the maximum number of zones is increased, chances are that the maximum number of nodes will have to be increased.

2. Identify all the arrays in the data structures which need to be expanded to accommodate the increase. At this time you can estimate the impact on SMART memory requirements. It may be necessary to limit size of the increase to keep the program within the space limitations of your computer.

3. Construct new COMMON block definitions for each of the affected data structures.

4. Insert the new definition into any module which references the structure(s), replacing the old definition, and re-compile those modules. The appendix "DATA STRUCTURE REFERENCES" identifies all modules so affected.

5. Locate any arrays local to a module which depend on a particular limit, and alter them. These are relatively rare in SMART; those which do exist are identified in this chapter.

6. Alter the BLOCK DATA subroutine, both to change the value of the limit and to specify initialization for the newly created array elements, if any.

The largest arbitrary network consists of 100 links, 50 nodes, and 100 zones. The largest ring/corridor structure consists of 8 rings and 6 corridors. Such a structure has 84 links, 43 nodes, and 85 zones. For the ring/corridor structure, we must always ensure that:

$$MXLINK \geq (MXRING - 1) * 2$$
$$MXNODE \geq (MXRING - 1) * MXCOR + 1$$
$$MXZONE \geq (MXRING - 1) * MXCDR * 2 + 1$$

Equality can hold for an exact ring/corridor structure; if additional zones, links, or nodes are to be added, forming a composite network, the limits must be raised. For example, if we have an eight-ring,

six-corridor structure, and two extra zones are to be added, MXZONE must be at least ((8-1)*6*2+1)+2 = 87.

MAXIMUM.NUMBER.OF.NODES (MXNODE) is the most important variable to examine because of its non-linear effect on storage requirements. There are three arrays whose size is proportional to the square of MXNODE. ORIGIN.DESTINATION.MATRIX (ODM) is dimensioned by the number of time periods and the maximum number of nodes squared, and is found in the DEMAND structure. LAST.AUTO.ROUTE.LINK (ROUTL) and AUTO.ROUTE.DISTANCE (ROUTD) are both dimensioned by the maximum number of nodes squared, and are found in the structure SHORTEST.ROUTES (ROUTES). The number of time periods is fixed at three; therefore, ODM contains 3 * MXNODE**2 elements, while ROUTL and ROUTD each contain MXNODE**2. The total number of elements involved here is therefore 5 * MXNODE**2. The current limit on MXNODE is 50. If this limit is raised from 50 to, for example, 51, (5 * 51**2) - (5 X 50**2) = 505 words will be added to SMART in these matrices alone. This is a large increase relative to the very minor increase in MXNODE. If we increase MXNODE by 50%, to 75, the total storage required for these arrays jumps from 12,500 words to 28,125 words, increasing SMART's total memory requirements by about 15%. Some other arrays are affected linearly, such as DEFAULT.NODE.IDENTIFIER (DNIDNT), which contains the default names of the nodes.

MAXIMUM.NUMBER.OF.MODES (MXMOD) is also critical to program size. It is important because nine of the ten arrays in RESULTS are dimensioned 3 by 3 by 6 by MXMOD. Therefore, increasing by 1 the maximum number of modes able to be analyzed in each run will require 486 additional words of storage for results. Data structure MODES.SELECTED (MODSEL) adds another 33 variables, largely through its modal parameter arrays such as COST.PER.VEHICLE.MILE (MCVM) and UNCONGESTED.LOCAL.SPEED (MSPEED). These modal parameters are passed to subroutine PRINT.MODAL.PARAMETERS (PMODES), where they are stored in the local array MPARM, which is dimensioned to 20 * MXMOD. Hence, when MXMOD is increased by 1, PMODES requires another 20 elements. Other local arrays bring the total additional requirement of core storage to 551 computer words. Thus, for small increases in limits, MXMOD is almost as critical as MXNODE in determining core requirements. Its importance relative to MXNODE decreases at large limit increases, since MXNODE is the only major limit with quadratic effect on memory.

The bounds on MXLINK and MXZONE are much less critical. Increasing the limit on MXLINK by 1 increases the number of array elements and words of storage by 80, with the greatest increase being contributed by data structure DEMAND through the arrays AUTO.TRAFFIC.VOLUME (TRAFIC), TRANSIT.TRAFFIC.VOLUME (TRANLD), TRANSIT.BOARDINGS (TRANON), and TRANSIT.DROPOFFS (TRANOF). These contribute 60 of the 80 additional elements. Other data structures affected by MXLINK include SHORTEST.ROUTES (ROUTES), URBAN.DEFAULT (UDFLT), URBAN.STRUCTURE (USTRUC), and a local array, XIMP, of the module GENERATE.SHORTEST.ROUTES (GROUTS). Because each link is assigned a default value in array DEFAULT.LINK.IDENTIFIER (DLIDNT), increasing MXLINK will require that DLIDNT be expanded and the additional links be initialized in the BLOCK DATA subroutine.

Increasing the limit on MXZONE by 1 contributes 36 additional words of storage to the program. The principal increase comes from the arrays RESIDENTIAL.VOLUME (VRES) and ACTIVITY.VOLUME (VACT) in data structure DEMAND, with each array requiring 6 new elements. Other data structures affected are DOOR.TO.DOOR.PARAMETERS (DORPRM), DEMAND.PARAMETERS (DPARM), MODES.SELECTED (MODSEL), URBAN.DEFAULT (UDEFLT), and URBAN.STRUCTURE (USTRUC). Local arrays affected are DMAX, EMPL, POP, TDIST, and ZONSEG in module GENERATE.DEMAND (GDEMND) and FEED in module DOOR.TO.DOOR (DOOR).

MAXIMUM.NUMBER.OF.RINGS (MXRING) and MAXIMUM.NUMBER.OF.CORRIDORS (MXCOR) have negligible direct effects on storage requirements. However, remember that an increase in either of these will probably require an increase in the number of links, nodes, and zones allowed, and these increases will have major effects. Extending the maximum number of rings by 1 adds 8 array elements, while extending the maximum number of corridors by 1 adds only 1 array element. Since all these array elements are single-precision real variables, they require the same increase in the number of words of storage. The data structures affected by MXRING include DEMAND.PARAMETERS (DPARM) and URBAN.DEFAULT (UDFLT). Only UDEFLT is affected by MXCOR. All affected arrays will require new initializations in the BLOCK DATA subroutine. Note that the RING.OUTER.RADIUS (RADIUS) and DEFAULT.OUTER.RADIUS (DRADUS) arrays in UDEFLT must be initialized identically. No local arrays are affected by these limits.

The above discussion shows that MXNODE is the most crucial factor in determining array storage requirements. From the viewpoint of storage, therefore, any increase in network detail should concentrate on increasing the number of links, zones, rings and corridors, and not on increasing the number of nodes. By the same token, decreasing storage requirements is more effective by decreasing the maximum number of nodes and modes analyzable in any one run than by decreasing any other limit.

5.3 MODULE REPLACEMENTS

Although any SMART module can be replaced by a module which performs an equivalent task, in most cases the task performed is so complex that complete replacement would be very expensive. The following modules, though, are relatively simple and/or isolated from the rest of SMART, and thus are prime candidates for replacement:

1. AGT.EFFECTIVE.VELOCITY (AGTVEL) computes the average speed of an AGT vehicle from the AGT system route spacing. The average speed includes slow-down and start-up time, but not any station dwell time. AGTVEL necessarily incorporates many assumptions about operating characteristics, such as motor torque, vehicle weight, top speed, and traction. This module can easily be replaced to incorporate either other assumptions or a different methodology. Be forewarned, however, that if the new methodology requires more information than is currently available to AGTVEL, other SMART code will have to be changed to maintain that information.

2. CONGESTED.LINK.SPEED (CLSPED) computes the resultant actual speed on a link from the link characteristics and the assumed transit mode share. Code relating to finding car-equivalents and number of non-diamond lanes will probably need to be retained, but the formula relating congestion to speed can easily be modified or replaced. Be sure that, no matter what happens, link speed never drops to zero.

3. CAPITAL.MAINTENANCE.ALLOCATION (CMALLO) incorporates rules for allocating daily charges to transit trips. While the rules used in SMART are meant to be fair (trips pay for excess capacity) they are not universal truths. Some people may wish to incorporate a different cost allocation scheme.

4. CONGESTED.ZONAL.SPEED (CZSPED) implements Smead's equation for finding the congested traffic speed in a CBD. Other formulations would require different versions of CZSPED.

5. TRANSIT.ROUTE (TRANRT) is perhaps the prime candidate for replacement. Its function is to decide on a transit route from one node to another node. The current algorithm is very simplistic. Note, however, that this module is called NNODE**2 times, and thus expensive code can get very expensive indeed.

   By route, we mean a series of link/direction/mode catagories. SMART has been carefully designed to accept almost any type of transit routing, with the following restrictions:

   > All patrons travelling from one node to another node must take the same transit route -- one cannot send one percentage one way and another percentage another way. Similarly, the route cannot change by time of day.
   >
   > Patrons returning take the same route in the reverse order.

   The procedure supplied with SMART follows the auto shortest route, using the highest-numbered available transit mode catagory (heavy rail in preference to light rail, light rail in preference to buses). This results in only one mode on each link getting any demand, and it also produces a lot of transfers.

5.4 MODAL ANALYSIS MODULES

Most of SMART is structured so that individual modes are handled in separate subroutines. Consequently, changes to the methodology are clearly located and isolated. These modules are:

1. CBD.AGT (CBDAGT)

2. CBD.AUTO (ACBATO)
3. CBD.CARPOOL (CBDCPL)
4. CBD.FIXED.ROUTE.BUS (CBDFRB)
5. DUMPER.AUTO (DPRATO)
6. DUMPER.CARPOOL (DPRCPL)
7. DUMPER.FIXED.ROUTE.BUS (DPRFRB)
8. FEEDER.AUTO (FDRATO)
9. FEEDER.CARPOOL (FDRCPL)
10. FEEDER.FLEXIBLE.ROUTE.BUS (FDRFLX)
11. FEEDER.FIXED.ROUTE.BUS (FDRFRB)
12. FEEDER.LIGHT.RAIL (FDRLRL) (Note that altering light rail to service an entire zone instead of just a corridor is a much more extensive change.)
13. FEEDER.PRIVATE.AUTO.TRANSIT (FDRPAT)

Any change to LINKER methodology is much more difficult, since actual analyses are divided among several LINKER and regionwide modules.

5.5 DEMAND GENERATION

SMART has been designed to facilitate incorporating alternative demand generation procedures. It is not even necessary to replace the default procedure. Merely code a subroutine which produces a series of calls to DEMAND.VOLUME (DEMVOL), and incorporate its name into the module GENERATE.DEMAND.REQUESTED (GENDEM). The procedure may make use of any available SMART data, but should alter nothing. Although the entire data structure DEMAND.PARAMETERS (DPARM) is available for passing parameters, problems may arise when different procedures require different parameters, and the default values for one are meaningless for another. Examine carefully the code for GENERATE.DEMAND (GDEMND), noting all of the dimensions of the problem. It is also possible to replace GDEMND with your own module, but that is not recommended.

5.6 LINKER EXPRESS STATUS DETERMINATION

LINKER chooses to run express (non-stop) buses on any route when the demand between the designated origin and destination nodes results in capacity-constrained service (buses are full). This is accomplished through the following statement in module LINKER:

```
EXPRES(TIME,DIR,MSI) = (DEM(OIR)*MHWMAX(MOD) .GE. MVCAP(MOD))
```

If the demand per minute in any direction multiplied by the maximum headway (which gives the load on the bus) is greater than the bus capacity, then -- for that period, direction and mode share -- EXPRES is .TRUE. and buses run non-stop. This rule is based solely on demand between the two designated nodes, even though patrons may be taking buses for only part of the route (buses in the first half, light rail in the other), and real buses might be able to pick up additional passengers traveling elsewhere. This rule was adopted for three principal reasons:

1. SMART does no vehicle routing. Thus it cannot know whether one link constitutes a continuation of another link, or a separate side-road.

2. In order to identify all travelers who could in theory benefit from non-stop service on the current route, SMART would have to look at all possible node pairs to determine which pairs might contribute to express bus loading. To control run costs, we have striven mightily to avoid execution code which runs as the square of the network size.

3. On a regionwide basis, the effects of running express buses is not significant. Such buses, while used more efficiently on the given route, cause inefficiencies in bus use on related routes.

The only easy way to change the rule for express bus determination is to operate express busses if the passenger load reaches some fixed proportion of capacity, say 80%. This will have little effect in most situations.

5.7 ADDING MODE TYPES

Adding a new FEEDER, DUMPER, or CBD mode type is a reasonable thing to attempt. Adding a new LINKER mode type is nearly impossible, and should not be attempted. Hence the following discussion will not apply to LINKER modes.

The folowing steps must be taken to add a new mode type for FEEDER, DUMPER, or CBD:

1. Increase by 1 the size of the arrays in MODE.DEFINITIONS (MODDEF). Then the initialization of these

arrays must be expanded to define default parameter values for the new mode type. MXDMOD initialization will have to be increased by 1.

2. A new mode number variable, similar to JAUTO, JCARPL, JFRB, etc. must be added to the MODES.SELECTED (MODSEL) structure, and initialized in BLOCK DATA to the default mode index of the new mode type.

3. Any modules which access either of the above data structures will have to have the structure declaration replaced, and the module recompiled. While this will include a great many SMART modules, it can, in most cases, be done automatically with a good text editor.

4. Any analysis package which is supposed to be able to model the new mode must be changed. Usually, this will require adding only one line of code, such as:

```
IF (MTYPE(MOD) .EQ. JSWIM)   CALL DPRSWM(...)
```

   and then writing a new modal subroutine.

5. Examine the subroutine PROCESS.MODES (PROMOD). This module handles updating modal parameters. Significantly, it contains some tests when particular parameters or values make no sense for a particular mode. You may wish to insert some test of this type.

No other changes should be necessary.

5.8 ADDING MODAL PARAMETERS

Oddly enough, it is more complicated in SMART to add new modal parameters than it is to add new mode types. This is because new parameters require new arrays, and new code to maintain those arrays, while new mode types simply require expanding existing arrays.

Currently, MODDEF and MODSEL contain arrays corresponding to known parameters. There is one array for the parameter vehicle speed (DMSPED in MODDEF and MSPEED in MODSEL), one array for the parameter vehicle capacity (DMVCAP in MODDEF and MVCAP in MODSEL), and so forth. Each new modal parameter that is defined will need new arrays declared in MODDEF and MODSEL. For example, we may have a new parameter NOISE. For this parameter, we may be define the arrays as DMNOIS in MODDEF and MNOISE in MODSEL.

The subroutine PRINT.MODAL.PARAMETERS (PMODES) contains an array MPARM which is dimensioned 20 by MXMOD. The first dimension gives the maximum number of modal parameters; the second gives the maximum number of modes. The first dimension size would have to be increased if more

than 20 parameters were to be used in SMART. SMART currently has 19 modal parameters, so that one new parameter can be added without changing the array. Other changes required in PMODES are changing the number of parameters NPARM and the number of labels NLBL in the DATA statement from 19 to the correct number of parameters, and assigning these additional parameters to MPARM in the DO loop terminated at statement number 2000.

NEW.MODE (NEWMOD) contains an assignment statement for each modal parameter, a statement which loads the current default value. Any additional parameters will have to have a new NEW.MODE statement.

PROCESS.MODES (PROMOD) contains the code to update modal parameter values depending on the user input. For a new modal parameter, a new option will need to be defined for the MODEPARM card, and PROMOD altered to process the option. Examine existing code to see how the new code should look and fit in. Note that PROMOD first looks for options with no trailing value needed, and then checks other options in the order of increasing restrictions on the allowable values. New code will have to go into this series in the correct place.

Naturally, any modal analysis routine which is supposed to incorporate the new parameter must be modified accordingly.

S.C.R.T.D. LIBRARY