NCS TIB 95-4

NATIONAL COMMUNICATIONS SYSTEM

TECHNICAL INFORMATION BULLETIN 95-4

TAMI MODEL
PROGRAMMER'S GUIDE
VOLUME I

JUNE 1995

OFFICE OF THE MANAGER
NATIONAL COMMUNICATIONS SYSTEM
701 SOUTH COURT HOUSE ROAD
ARLINGTON, VA 22204-2198

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1995	Final Report

4. TITLE AND SUBTITLE

TAMI Model Programmer's Guide Volume I

5. FUNDING NUMBERS

DCA100-91-C-0015

6. AUTHOR(S)

Andre Rausch

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Booz, Allen & Hamilton, Inc.
8283 Greensboro Drive
McLean, Virginia 22102

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Communications System
Office of Technology and Standards Division
701 South Court House Road
Arlington, Virginia 22204-2198

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NCS TIB #95-4
VOL I

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

As part of the ongoing effort to analyze the performance of the public switched network (PSN) and the programs of the National Level National Security and Emergency Preparedness (NS/EP) Telecommunications Program, Office of Technology and Standards Division (N6) has developed a number of computer-based models. The Traffic Analysis by Method of Iteration (TAMI) model was developed to measure the effects of telecommunications traffice congestion in stressed local and long distance networks. This document provides the first of two volumes of the TAMI Programmer's Manual. Together these volumes provide the software description necessary for a programmer to support future maintenance and enhancements to the TAMI model. The TAMI User's Manual provides the information for users who wish to operate the model. Volume I documents the first 11 of 23 modules that form the TAMI model. It is assumed that the reader has a basic understanding of the PSN and working knowledge of the architectures of the three major inter-exchange carrier networks (IEC) and the local exchange carrier (LEC). A programmer using TAMI should have a working knowledge of the UNIX operating system and the 'C' and FORTRAN programming languages. Background in voice teletraffic engineering and analytical modeling and simulation is desirable.

14. SUBJECT TERMS

Inter-Exchange Carrier (IEC)
Local Exchange Carrier (LEC)
Public Switched Network (PSN)

15. NUMBER OF PAGES

130

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASS	UNCLASS	UNCLASS	UNLIMITED

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE - See authorities.
NASA - See Handbook NHB 2200.2.
NTIS - Leave blank.

Block 12b. Distribution Code.

DOD - Leave blank.
DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA - Leave blank.
NTIS - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

NCS TIB 95-4

NATIONAL COMMUNICATIONS SYSTEM

TECHNICAL INFORMATION BULLETIN 95-4

TAMI MODEL
PROGRAMMER'S GUIDE
VOLUME I

JUNE 1995

OFFICE OF THE MANAGER
NATIONAL COMMUNICATIONS SYSTEM
701 SOUTH COURT HOUSE ROAD
ARLINGTON, VA 22204-2198

NCS TECHNICAL INFORMATION BULLETIN 95-4

TAMI MODEL
PROGRAMMER'S GUIDE
VOLUME I

JUNE 1995

PROJECT OFFICER

ANDRE RAUSCH
Electronics Engineer
Office of Technology
and Standards

APPROVED FOR PUBLICATION:

Dennis Bodson

DENNIS BODSON
Assistant Manager
Office of Technology
and Standards

FOREWORD

The National Communications System (NCS) is an organization of the Federal Government whose membership is comprised of 23 Government entities. Its mission is to assist the President, National Security Council, Office of Science and Technology Policy, and Office of Management and Budget in:

- The exercise of their wartime and non-wartime emergency functions and their planning and oversight responsibilities.

- The coordination of the planning for and provisions of National Security/Emergency Preparedness communications for the Federal Government under all circumstances including crisis or emergency.

In support of this mission, the NCS has conducted studies and analyses to assess the potential for serious damage to portions of the Nation's telecommunications infrastructure due to various threats. The purpose of the work is to provide guidance to programmers on the TAMI module structure.

Comments on this TIB are welcome and should be addressed to:

Office of the Manager
National Communications System
Attn: NC-TS
701 S. Court House Road
Arlington, VA 22204-2198

TABLE OF CONTENTS

TAMI Model Programmer's Guide, Part I

1.0 Introduction

The Office of the Manager, National Communications System (OMNCS) Office of Technology and Standards (NT) is responsible for a broad range of initiatives including Federal telecommunications standards development, network performance analyses, and technology review. As part of the ongoing effort to analyze the performance of the public switched network (PSN) and the programs of the National Level National Security and Emergency Preparedness (NS/EP) Telecommunications Program (NLP), NT has developed a number of computer-based models. Most recently, the Traffic Analysis by Method of Iteration (TAMI) model was developed to measure the effects of telecommunications traffic congestion in stressed local and long distance networks.

1.1 Purpose

This document provides the first of two volumes of the TAMI Programmer's Manual. Together, these volumes provide the software description necessary for a programmer to support future maintenance and enhancements to the TAMI model. A separate document, the TAMI User's Manual, provides the information necessary for users who wish to operate the model.

1.2 Scope

Volume I of the TAMI Programmer's Manual documents the first 11 of 23 modules that form the TAMI model. It is assumed that the reader has a basic understanding of the PSN and a working knowledge of the architectures of the three major inter-exchange carrier networks (IEC) and the local exchange carrier networks (LEC). Furthermore, a programmer using TAMI should have a working knowledge of the UNIX operating system and the 'C' and FORTRAN programming languages. A background in voice teletraffic engineering and analytical modeling and simulation is desirable to understand the algorithmic details of the TAMI model. The TAMI programmer will find it useful to be familiar with the references provided at the end of this document, which describe previous TAMI analyses, modeling concepts, algorithms, and programmer's manuals of related software.

1.3 Background

The nation's PSNs continue to be a focus of NCS modeling efforts because these networks comprise the largest, most diverse set of telecommunications assets in the United States. Furthermore, the NCS directs its NS/EP telecommunications enhancement activities toward the PSN. Additionally, most NCS member organizations rely on the PSN for conducting their NS/EP responsibilities.

The NCS has moved to measuring network performance using call completion probability in addition to connectivity because this approach captures the effects of traffic congestion. Traffic congestion is prevalent during many of the national emergencies and disasters of concern to the OMNCS.

In support of PSN traffic congestion analyses, NT has developed the TAMI model. This model is only intended for use in networks stressed by physical damageand/or traffic overload. This model measures congestion in the combined local and long-distance networks of the PSN. TAMI evaluates congestion for ordinary telephone users and for NS/EP users who benefit from planned or existing NLP enhancements. In addition to measuring nationwide congestion, the TAMI model has been expanded to model regional congestion caused by focused overloads. Focused overloads are common during events that only affect part of the country, such as earthquakes and hurricanes, during which the affected region may be subject to unusually high volumes of traffic originating from the rest of the country. As the TAMI model continues to evolve, it provides a more accurate tool for understanding the effects of congestion in the PSN.

The TAMI model has been used by both NT and the Office of Plans and Programs (NP) to measure congestion in the PSN subject to damage from electromagnetic pulse (EMP), fallout radiation, nuclear blast scenarios, and, more recently, earthquakes. An analysis has successfully been conducted to determine the sensitivity of the model's network performance results to network management and engineering assumptions made in the absence of complete PSN data. In view of continued plans to employ and enhance the TAMI model, this document provides the first of two Programmer's Manual volumes. These volumes will be supplemented by a User's Manual.

1.4 Organization

This report is organized into three sections. Section 1.0 provides an introduction, describing the purpose, scope, background, and organization.

Section 2.0 provides a high level overview of the TAMI model and describes the interrelationships, data flow, and interfaces among of the eleven software modules encompassed by this report.

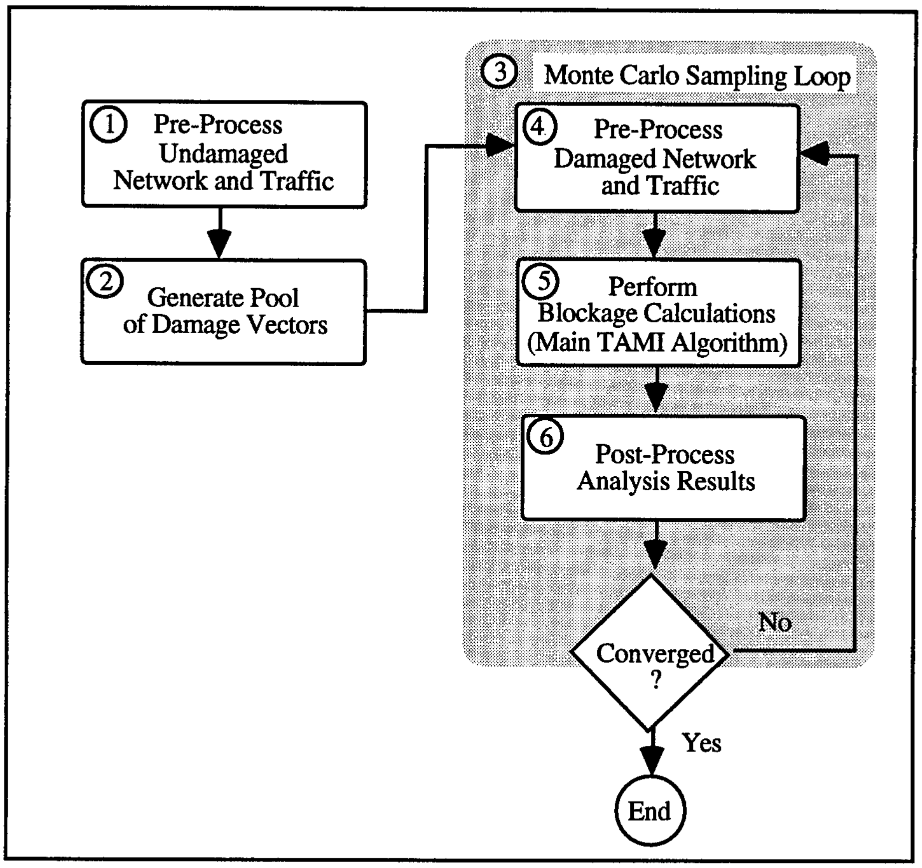
Section 3.0 contains detailed documentation of the first 11 TAMI software modules and each module's component functions.

2.0 High Level TAMI Description

This section provides a high level overview of the TAMI Model from a programming viewpoint. A number of NCS reports already exist which describe the TAMI algorithms, assumptions, and modeling techniques (References 2, 3, 5, 6). The purpose of this overview is to focus on the interrelationships, data flow, and interfaces among the software modules that constitute the TAMI model.

The TAMI model can be divided into six main functional processes, depicted in Exhibit 2.1. Each of these processes operate on both IEC and LEC data and can be described at more detailed levels.

Exhibit 2.1
TAMI High-Level Flow Diagram



In addition to QTCM, which has been previously documented as a stand-alone model, there are 23 TAMI modules totaling an estimated 30,000 lines of code. Exhibit 2.2 identifies each of these modules, categorized by the six functional areas above. It also provides the approximate lines of code for each module and indicates whether it appears in Volume I or Volume II of the TAMI Model Programmer's Manual. As shown, Volume I encompasses the pre-processing of the undamaged IEC networks and the generation of sampling pools of EMP damage vectors. These two functional areas are discussed in more detail in Sections 2.1 and 2.2 respectively. Exhibits 2.3 and 2.4 provide a diagrammatic road map to these sections, depicting the overall data flow for AT&T, MCI, and Sprint network data through the Volume I modules.

Exhibit 2.2
Table of TAMI Modules

Functional Area Within TAMI	Module Name	Approx Lines of Code	Vol I	Vol II
Pre-Process Undamaged Network and Traffic				
IEC Networks	cg_make	200	✓	
	span_make	300	✓	
	array_make	200	✓	
	mk_ncam_path	4800*	✓	
	rem_dup	200	✓	
	sort_path	130	✓	
	3ch_4ch	200	✓	
	match_trunk	830	✓	
	mkpath	275	✓	
LEC Networks and End-to-End Traffic Matrix	attlive	900		✓
	mcilive	1300		✓
	sprlive	1300		✓
Generate Pool of Damage Vectors	damage	1600	✓	
	mklink	500	✓	
Monte Carlo Sampling Loop	tami	650		✓
Pre-Process Damaged Network and Traffic				
IEC Networks	mkrout	700		✓
	qtrans_gen	800		✓
LEC Networks and End-to-End Traffic Matrix	attwdmg	2300		✓
	mciwdmg	2300		✓
	sprwdmg	2300		✓
	merge	850		✓
Perform Blockage Calculations				
LEC Networks	lecam	4500		✓
IEC Networks	qtcm	N/A	N/A	N/A
Post-Process Analysis Results	keepstats	300		✓

* Includes code linked from IDA/CAM model, Reference 4

Exhibit 2.3
AT&T and Sprint Data Flow Through Volume I TAMI Modules

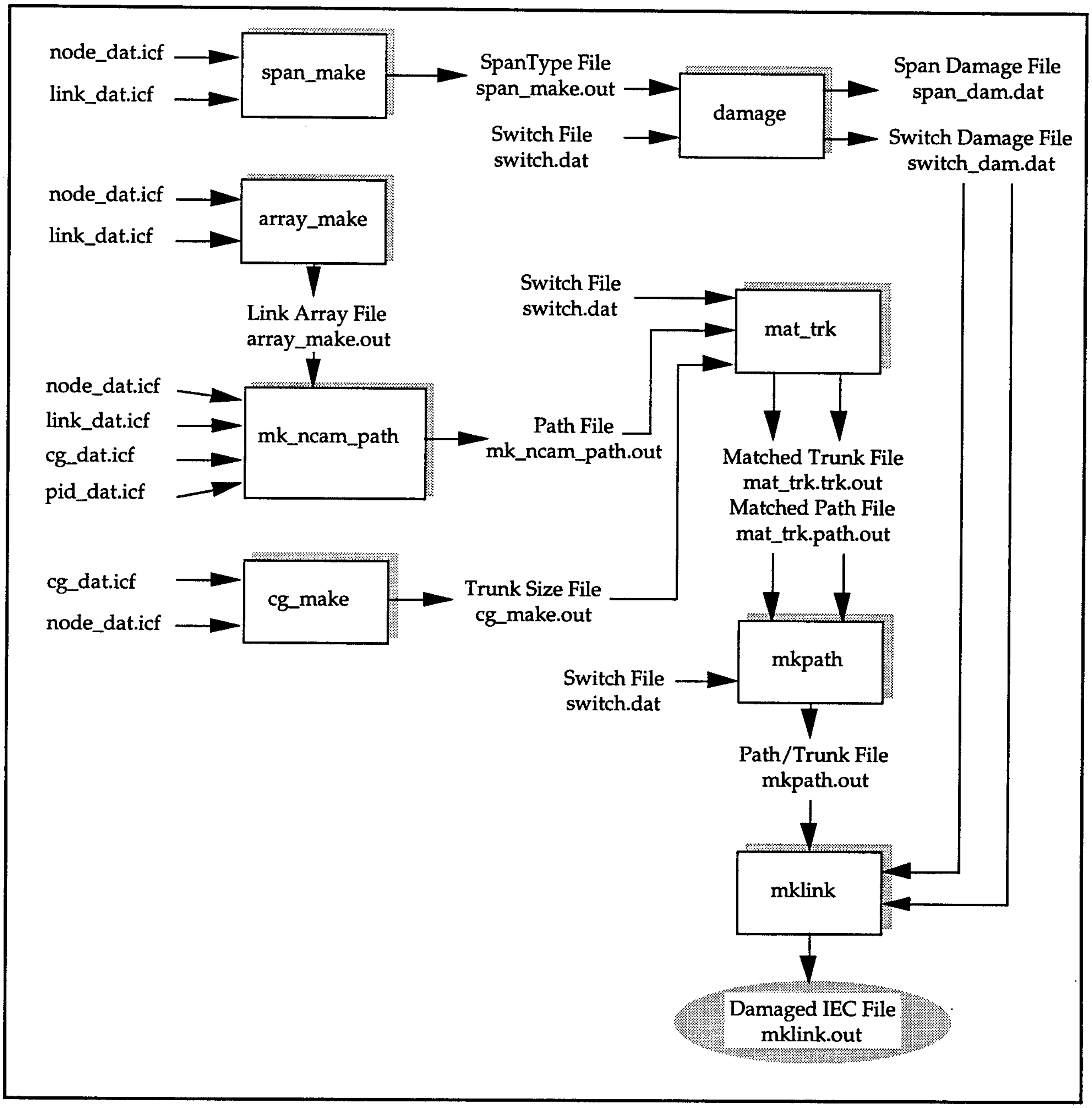
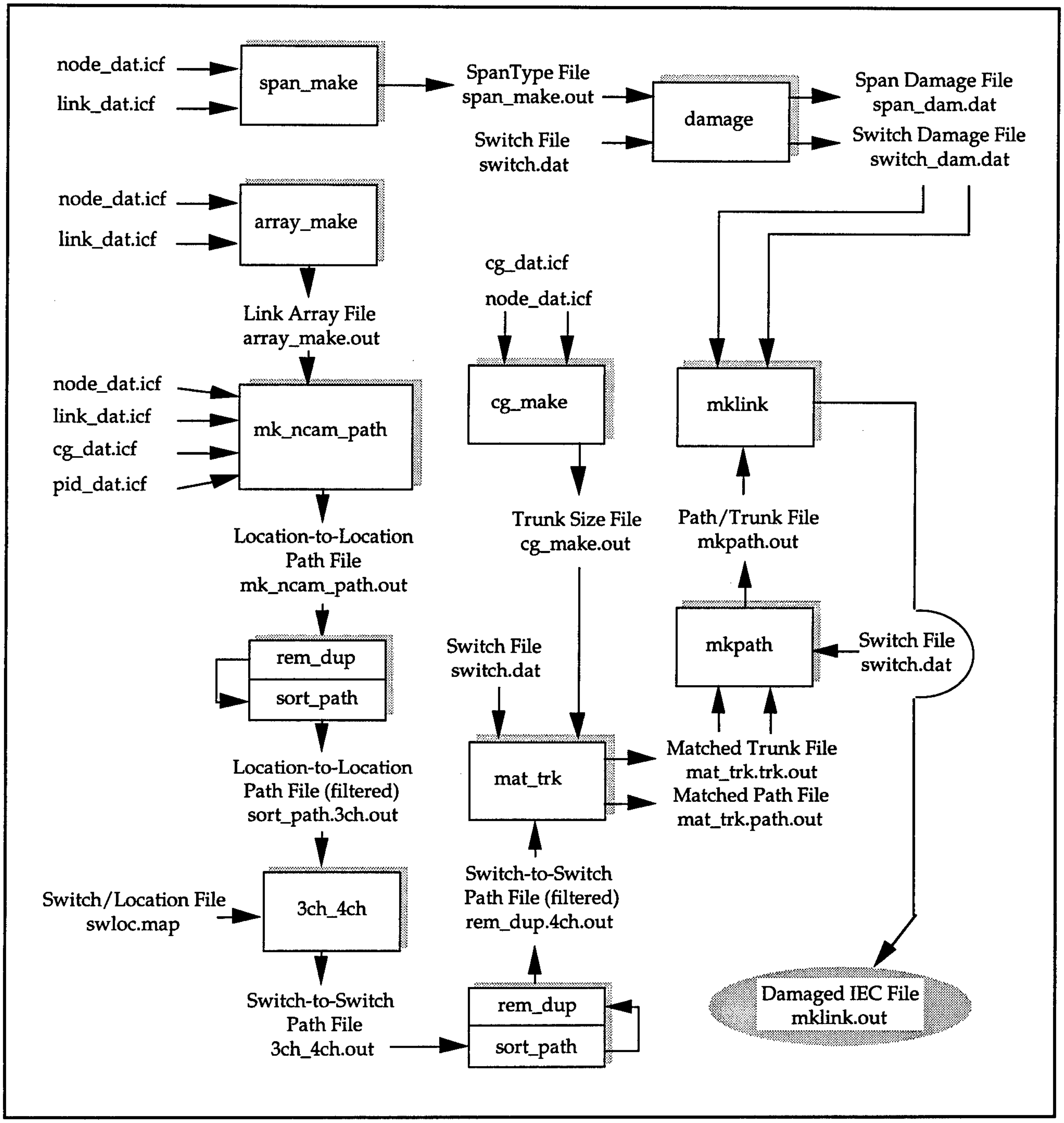


Exhibit 2.4
MCI Data Flow Through Volume I TAMI Modules



2.1 Pre-processing the Undamaged IEC Networks

There are three goals to this stage: (1) to reformat the IEC data from ICF format (see Appendix A) into a format usable by TAMI, (2) to address data anomalies or gaps, and (3) to produce a mapping between the physical and logical IEC network descriptions. This section describes each of the nine modules used to perform the pre-processing objectives on the undamaged IEC networks. Because each IEC data set has unique anomalies, some modules provide the option to specify the IEC or were designed to be run only on a specific carrier's data.

The first four modules, *cg_make*, *span_make*, *array_make*, and *mk_ncam_path*, are designed to reformat the ICF files so they can be used by TAMI. Each of these modules is generic (i.e. runs the same on each IEC) and uses hard coded input and output file names. The four ICF input files are as follows (see Appendix A for further description):

node_dat.icf	describes all IEC network nodes, including switches, repeaters, etc.
link_dat.icf	describes the links (or spans) between nodes
cg_dat.icf	describes the logical circuit groups (trunk groups) in the IEC backbone
pid_dat.icf	identifies the physical transmission paths (chains of spans) that connect the backbone switches

Each of these four modules is described:

cg_make: This module uses the cg_dat.icf and node_dat.icf files to decode the ICF's numerical circuit group format into an easy-to-read trunk group file based on standard CLLI codes. It outputs a file called 'cg_make.out.' For Sprint only, the circuit groups in the 1993 ICF files were maintained as DS1s. A standard UNIX utility called 'nawk' was used to multiply to trunk size field by 24 to convert to DS0s.

span_make: This module uses the node_dat.icf and link_dat.icf files to produce a span type file in the format traditionally used for NCAM EMP damage. Span types, such as fiber optic, microwave, and T1, are converted from the codes used in ICF to the equipment type codes used for EMP damage. *Span_make* produces the output file 'span_make.out.'

array_make: This module almost the same as *span_make*, except that the output span file it produces is stored in an indexed numeric format in a binary file. This output file, 'array_make.out,' is read by *mk_ncam_path* to assist it in building paths from spans.

mk_ncam_path: This module uses all four ICF files as well as the output file from *array_make* to build a physical transmission path file in the format used by TAMI. This output file is called 'mk_ncam_path.out.'

The next three modules, *rem_dup*, *sort_path*, and *3ch_4ch*, were designed specifically to address anomalies in MCI's 1993 path data. Each of these modules performs a specific data filtering step on the 'mk_ncam_path.out' file as described below:

3ch_4ch: MCI's 1993 path data does not use 4 character switch codes as endpoints, but uses 3 character location codes. (This characteristic has been changed in more recent versions of the data, so *3ch_4ch* may not be needed in the future.) Because some locations house more than one switch, a path that ends at such a location implies a multiplicity of paths to each switch in the building. This module addresses this data anomaly by mapping each location-to-location path record to all possible combinations of switch-to-switch path records. In addition to the location-to-location input path file, it uses a list of 4 character switches and corresponding 3 character location codes to produce the output switch-to-switch path file. All of these files have user-specified names.

rem_dup: MCI's 1993 path data contains a number of duplicate records which needlessly enlarge the size of the 'mk_ncam_path.out' file. *Rem_dup* compares each path record to the previous two records to remove adjacent duplicates. The resulting output file has a user-specified name (usually called 'rem_dup.out'), and is used as the filtered path file for subsequent steps. To fully prevent the possibility of duplicate records, this module is run on both the location-to-location path file (prior to *3ch_4ch*) and on the switch-to-switch path file output by *3ch_4ch*.

sort_path: This module sorts the records in the path file alphabetically by the CLLI codes of the originating and terminating path endpoints in order to group paths between the same switches. Subsequent modules expect the path file to be ordered in this manner. The resulting output file has a user-specified name (usually called 'sort_path.out'), and is used as the filtered path file for subsequent steps. *Sort_path* is run together with *rem_dup* on both the location-to-location path file (prior to *3ch_4ch*) and on the switch-to-switch path file output by *3ch_4ch*.

The final two modules, *mat_trk* and *mkpath*, perform the task of mapping the logical trunk groups to the physical transmission paths. The final output file combines logical and physical network data to describe the number of trunk groups per path. This trunk group per path breakdown facilitates the later task of determining the impact of a damaged path on the overall trunk group capacity between two switches. In this manner, physical damage can be correlated to reduced network capacity. *Mat_trk* and *mkpath* are described as follows:

mat_trk: This module divides trunk group quantities among the physical paths. For example, if there are 72 trunks (3 DS1s) in the trunk group from switch A to switch B, and three distinct physical paths between A and B, *mat_trk* assigns 24 trunks to each physical path. Each record in the output trunk group file will therefore have a one-to-one correspondence with paths in the output path file. *Mat_trk* also checks path and trunk group endpoints against a list of backbone switches to make sure that only paths and trunk groups that originate and terminate within the toll network are used. *Mat_trk* addresses cases where there are mismatches between the trunk group file and the path file–where either a trunk group exists without a corresponding path, or a path exists without a corresponding trunk group.

mkpath: The *mkpath* module combines each record of the *mat_trk* output trunk file to the corresponding record in the *mat_trk* output path file. In the process, it replaces the CLLI codes of the endpoint switches with index numbers into the switch list. This step optimizes future processing tasks. The output file is a combined and indexed trunk group/path file, given a user-specified name.

2.2 Generating Damage Vectors

A damage vector can be thought of as one random instance, or scenario, of possible network damage. Given the probabilities of failure of each network component, such as switches and spans (including endpoint nodes such as repeaters), it is possible to generate many different damage vectors, where each network component is identified as being in either the damaged or undamaged state. Since each damage vector is based on a common set of probabilities, they will all tend to have similar average levels of damage. However, each vector represents a slightly different, random outcome that could occur.

As described in previous TAMI analyses (References 2, 3, 5), TAMI uses a Monte Carlo approach, sampling as many damage vectors as needed to reach a suitable confidence level in overall network performance results. This section describes the two modules, *damage* and *mklink*, that generate the pool of damage vectors from which the modules in the Monte Carlo loop sample. *Damage* is generic to both LEC and IEC spans and nodes, whereas *mklink* is specific to IECs–it maps the effect of damaged component spans and nodes to the long-haul transmission paths in these networks. While the specific version of *damage* described in this document calculates telecommunications damage due to EMP and fallout radiation, future versions of this module may be developed to characterize damage due to other threats, such as earthquakes, floods, or hurricanes.

The modules *damage* and *mklink* are described as follows:

damage: This module generates a user-defined number of randomly sampled damage vectors for two general categories of assets: nodes and spans. Damage is based on the susceptibility of each equipment type to EMP or fallout radiation. Each span and node equipment type has a cumulative distribution function (CDF) which defines the equipment's probability of failure. The input span or node files must have a field describing equipment type. In the output span or node file, this field will be replaced with the string of damage vectors–one 0 or 1 for each damage vector requested. It is most common to generate a pool of 15 damage vectors for each type of damage (low, medium, and high EMP intensity).

mklink: The *mklink* module generates pools of damaged IEC paths based on the component span and node damage files produced by *damage*. It uses the combined trunk/path file and the span and node damage files as input. It steps through the endpoint nodes and the chain of spans that make up each path, checking each component to determine if the overall path is damaged or surviving. The path's list of spans are replaced by the string of damage vectors in the output file.

3.0 Module Descriptions

This section describes the first 11 TAMI modules. Overall documentation is provided for each module, followed by detailed documentation of each component function. Secion 3.1 describes the documentation conventions used in the sections that follow. Many of these conventions were adopted to make the documentation independent of the details of 'C' syntax

3.1 Documentation Conventions

This manual documents modules and functions coded in the 'C' programming language. Throughout the manual certain conventions which may differ slightly from standard 'C' terminology have been adopted in order to more clearly describe data types, inputs, outputs, includes and file types . In addition the `courier` font is used to denote module elements such as variable names, file names, call syntax, etc....

Variable names are defined by the following conventions:

Convention	Example	Definition
character	`c`	The standard 'C' data type *char*
integer	`i`	The standard 'C' data type *int*
float	`real`	The standard 'C' data type *float*
long integer	`pos`	The standard 'C' data type *long*
file	`outfile`	The standard 'C' data type *FILE*, a pointer to a file string
extern	`optarg`	The standard 'C' data type modifier *extern* indicating the variable is declared outside the module (e.g., the operating system)
global	`idx_num`	Variables that are declared globally accessible from any function
double	`supp_cdf_table[][]`	The standard 'C' *double* precision float data type
pointer	`*varname`	Denotes a pointer to any variable, *varname*
Constants	`PATH_REC`	The standard 'C' *#define*
structure	`p_struct    p[]` *with field* integer `p[].three`	The standard 'C' aggregate, heterogeneous hierarchical data structure composed of a main variable name and sub fields of multiple data types

Inputs/Outputs are defined by the following conventions:

Inputs are of two types: 1) formal inputs are passed in by the calling function; 2) global inputs are variables as defined above.

Outputs are of three types: 1) the formal parameter is returned tby he called function; 2) arguments that are passed by reference are modified; 3) global outputs are variables as defined above

Includes are defined by the following conventions:

Includes are of two types: 1) Standard 'C' defined function sets, e.g., `<stdio.c>`; and 2) user defined function sets, e.g., `"fileio.c"` (see Appendix B)

File formats are defined by the following conventions:

1) <CLLIA>, <CLLIZ>, <size>
2) (c11, 1x, c11, 1x, i6)

Line1 shows the names and relative positions of the fields within each record. Line 2 shows the data type and length of each field, where:

c=character
x=space
i=integer

f=float

Placement of algorithm and variables local to `main()`:

Module level descriptions are inclusive of the algorithms and variables local to the function `main()` for each module

Equality of trunk group and circuit group

Throughout this document the terms trunk group and circuit group are used interchangeably

3.2 cg_make: Make Circuit Groups Module

Purpose This module uses the cg and node ICF data files to produce a trunk group formatted for use with other TAMI modules.

Call Syntax `cg_make`

Input Files

ICF node file	This file name is hard-coded to `node_dat.icf.` See the ICF file description in Appendix A for more information regarding contents and format.
ICF cg file	This file name is hard-coded to 'cg_dat.icf.' See the ICF file description in Appendix A for more information regarding contents and format.

Output Files

trunk size file	This file contains a list of trunks. Each line contains two 11-character CLLI codes (the two span endpoints), along with an associated trunk size .
format	<CLLI A>, <CLLI Z>, <trunk_size> (c11, 1x, c11, 1x, i6)
example	ADMSTX0101T AKRNOH2505T 120

Includes

`<stdio.h>`	
`<string.h>`	
`"icf.h1"`	Defines constants used to process ICF files
`"fileio.c"`	User-defined I/O functions; see Appendix B

Constants Constants used in `array_make` are defined in `"icf.h1"` as follows:

NODE_MAX	64	number of characters in a ICF node file record
CG_MAX	38	number of characters in a ICF cg file record

Global Variables none

Local Variables Variables local to `main()`: none.

Component Functions

`make_cgs()`	builds output trunk group records by looping through each record in the cg file
`node_find()`	returns the CLLI code for a given node index
`fget()`	user defined utility I/O function; see Appendix B

Function Tree

```
main() — make_cgs() ─┬─ fget()
                     └─ node_find() — fget()
```

Algorithmic Description

The purpose of this module is to produce a trunk group file formatted for input to other TAMI modules. `Cg_make()` performs this task by replacing the node index numbers in the cg records with the node CLLI codes and printing out these endpoints along with a trunk group quantity..

This module consists of a call to the `make_cgs()` function, which reads in the input files, performs the amin algorithm and writes the output trunk group file.

Inputs none; operates on global variables

Outputs

file	`*writefile`	pointer to the output trunk group, hardcoded to `'cg_make.out'`
returns	none	

Purpose This function reformats information from the ICF node and cg files to create the output trunk size file.

Called By `main()`

Calls To `make_link(), fget()`

Local Variables

file	`*nodefile, *cgfile`	pointers to the input ICF node and cg data files
	`*writefile`	pointer to the output trunk size file
integer	`no_nodes`	the number of records in the ICF node file
	`idx_num`	loop count variable for each cg record
	`file_pos`	pointer to a particular byte or position in the ICF cg file
	`status`	not used
	`no_cgs`	the number of records in the ICF cg file
	`nodeAidx`	the node index of the originating circuit group endpoint
	`nodeZidx`	the node index of the terminating circuit group endpoint
	`ntrkqty`	the quantitiy of trunks in a circuit group record
character	`clli_A[]`	used to hold the originating node CLLI code
	`clli_Z[]`	used to hold the terminating node CLLI code
	`node_info[]`	used to hold a line/record from the ICF node file
	`cg_info[]`	used to hold a line/record from the ICF cg file

Global Variables none

Algorithmic Description

The function begins by opening the input and output files and counting the number of records in the cg file. For each record in the ICF circuit group file, this function parses the circuit group endpoints (denoted by node indexex) and the circuit group quantity fields. For the node index endpoints, `node_find()` is called, which returns the node CLLI code. `Make_cgs()` writes the CLLI code endpoints and trunk group size to the output file. When each input CG file record has been reformatted, the function returns.

Inputs

integer	idx_num	the index of a node in the ICF node file
file	*nodefile	the file pointer to the ICF node file

Outputs

character	*clli_ptr[]	a pointer to a string in which a node CLLI code is returned
returns	not used	

Purpose Given a node index, this function returns the node CLLI code

Called By make_cgs()

Calls To fget()

Local Variables

character	node_info[]	array to hold node data
	clli_temp[]	array to temporarily hold a CLLI code
long	file_pos	position in the ICF node file

Global Variables none

Global Constants NODE_MAX the number of characters in a node file record

Algorithmic Description

Given a node index, this file calculates the node record's position in the node CLLI code.

This function begins by using the node index to calculate the node record's position in the ICF node file (an extra record must be skipped to account for a header record). Then, this function calls `fget()` in order to read node record from the nodefile into the buffer `node_info[]`. The node CLLI code field is then parsed and checked to make sure it is not a dummy code (all X's: "XXXXXXXXXXX"). If the CLLI code is valid, it is assigned to `clli_ptr` to be returned to the calling function

3.3 span_make: Make Spans Module

Purpose This module decodes span information from the node and link ICF data files; the span output file is used by the `damage` module, and is referenced by span record indices in the path file.

Call Syntax `span_make`

Input Files

ICF node file — This file name is hard-coded to 'node_dat.icf.' See the ICF file description (Appendix A) for more information regarding contents and format.

ICF link file — This file name is hard-coded to 'link_dat.icf.' See the ICF file description (Appendix A) for more information regarding contents and format.

Output Files

span file — This file contains a list of spans. Each line contains two 11-character CLLI codes (the two span endpoints), a 2-character equipment code, and Vertical-Horizontal coordinates for each endpoint.

format — <CLLI A>, <CLLI Z>, <equipment code>, <V-coord A>, <H-coord A>, <V-coord Z>, <H-coord Z>
(c11, 1x, c11, 1x, c2, 1x, i5, 1x, i5, 1x, i5, 1x, i5)

Includes

`<stdio.h>`	
`"icf.h1"`	Defines constants used to process ICF files
`"fileio.c"`	User-defined I/O functions; see Appendix B

Constants Constants used in `span_make` are defined in `"icf.h1"` as follows:

`NODE_MAX 64`	number of characters in a ICF node file record
`Node_idx_fldlen 4`	number of characters in the node index field of the ICF node file
`LINK_MAX 24`	number of characters in a ICF link file record
`CLLI_SIZE 12`	number of characters in a CLLI code (11) plus the standard 'C' null character

Global Variables none

Local Variables Variables local to `main()`: none.

Component Functions

`make_spans()`	loops through the node file to build the data records
`make_link()`	for a given node, makes a record for each node link, given in the link file
`node_find()`	finds the corresponding node index in the link file for each node index in the node file
`return_type()`	reclassifies equipment type
`fget()`	reads in a given number of bytes from a given file position

Function Tree

```
main() —make_spans() ┬─ fget()
                     └─ make_link() ──┬─ fget()
                                      ├─ node_find() ── fget()
                                      └─ return_type()
```

Algorithmic Description

The purpose of this module is to decode span information from the node and link ICF data files. Span/link records are the shortest identifiable segments of telecommunications transmission networks, typically representing segments between repeaters, or between a repeater and multiplexing/switching equipment. The `span_make` module outputs a list of these spans, designated by the node indices of the two span endpoints, along with corresponding equipment types and vertical/horizontal coordinates.

This module uses the `span_make()` function to open the input/output file, and to control the building of the records to be written to the output file. It then calls `make_link()` (which in turn calls `node_find()` to set an index between the node and link files, and `return_type()` to reclassify equipment types). `Make_link()`, assembles the completed records, and writes them to the output file.

Inputs none

Outputs

file `*writefile` pointer to the output span file

returns integer number of nodes in ICF node file for which links have been processed in the ICF link file

Purpose For each record in the ICF node file, this function builds the spans that terminate at that node from information contained in the ICF link file, and builds records with these spans and their associated equipment codes and vertical/horizontal coordinates, .

Called By `main()`

Calls To `make_link(), fget()`

Local Variables

file	`*nodefile, *linkfile`	pointers to the input node and link ICF data files
	`*writefile`	pointer to combined node/link/equipment code output file
integer	`link_head`	reference to the beginning of a node's ICF link file references
	`link_tail`	reference to the end of a node's ICF link file references
	`no_nodes`	the number of records in the ICF node file
	`idx_num`	general loop count variable
	`file_pos`	pointer to a particular byte or position in the ICF node file
	`status`	non-zero if the function's call to `make_link()` was successful
	`v_temp`	temporarily holds the vertical coordinate of the current node
	`h_temp`	temporarily holds the horizontal coordinate of the current node
character	`clli_temp[]`	used to hold a node CLLI code
	`node_info[]`	used to hold a line/record from the ICF node file

Global Variables file `node_dat.icf`
`link_dat.icf`

Algorithmic Description

This function processes the ICF node file, record by record. For each node record, it parses the node CLLI code field and makes sure it is not a dummy code (all X's: "XXXXXXXXXXX"). Then it parses the `link_head` and `link_tail` fields from the node record. These fields point to the node's corresponding ICF link file records. If the `link_head` and `link_tail` fields are valid (non-zero), then `make_link()` is called, which actually steps through each of the node's link records , reclassifies equipment types, captures vertical and horizontal coordinates and builds the spans. If the call to `make_link()` returns a `status` of `0`, then a general error message is printed to the output terminal.

Inputs

integer	head_pt	the pointer to the first link file record describing the current node's links
	tail_pt	the pointer to the last link file record describing the current node's links
	node_v	vertical coordinate of the current node
	node_h	horizontal coordinate of the current node
file	*link_file	the pointer to the ICF link file
	*node_file	the pointer to the ICF node file
	*outfile	the pointer to the span output file
character	node_clli[]	the current node CLLI code

Outputs

returns	integer	a status indicator, equal to 1 if make_link() ran without error, or 0 if an error occurred in make_link()

Purpose Given a node index number and the starting and ending link records for that node, this function builds the spans that terminate at that node and writes them to the output file, along with the span equipment type and V-H coordinates of the span endpoints.

Called By make_spans()

Calls To fget(), node_find(), return_type()

Local Variables

character	linkrecord[LINK_MAX+1]	used to read in and hold a record from the link file
	link_clli[]	holds the CLLI code of the terminating span endpoint
	link_type[]	holds the TAMI span equipment type
	type	holds the ICF span equipment type
long	f_pos	pointer to the current byte/position in the ICF link file
integer	diff	the number of link records to be processed for a given node, equal to the difference between tail_pt and head_pt
	i	refers to a position in linkrecord[] as it is sequentially processed
	j	counts the number of link records processed, up to diff
	status	equal to 1 if make_link() ran without error, 0 if an error occurred
	link_v, link_h	the V-H coordinates of the terminating span endpoint
	node	the index of a span endpoint in the link file

Global Variables none

Algorithmic Description

In the ICF file format, a node file record points to the range of link file records that describes the other endpoint of links that use that node (see Appendix A). Each link file record has fields reserved for up to four such link endpoints. For a given node, starting link record, and ending link record, function make_link() builds each link/span along with an equipment code, and V-H coordinates of each endpoint and writes this data to the output file. If an error occurs, make_link() returns a status of 0.

The function starts by calculating diff, the number of link records to process from head_pt to tail_pt. It then enters a loop to read each of the diff link records into linkrecord[] to reclassify the equipment code, to look up the CLLI codes and V-H coordinates of the span endpoints and to verify that no file read error occurred. For each link record, the function scans link endpoint fields until all four link endpoints have been processed or a blank field is encountered. After all of the cuurent node's span/link records have been processed, the function returns the value of status to the calling routine.

Inputs

integer	idx_num	the index of the node being constructed in this module
	*vc	pointer to the vertical coordinate of the node
	*hc	pointer the horizontal coordinate of the node
file	*nodefile	the file pointer to the ICF nodefile
character	clli_temp[]	temporarily holds the CLLI code

Outputs

returns: no formal values are returned; outputs are written to the variable address for the vertical and horizontal coordinates

Purpose: To look up a node by index and return the CLLI code and vertical and horizontal coordinates and node.

Called By: make_link()

Calls To: fget()

Local Variables

character	node_info[]	string to hold a node record
long	file_pos	position in the ICF node file

Global Variables: none

Global Constants: NODE_MAX number of characters in an ICF node file record

Algorithmic Description

This function is called by mk_link() to process the nodefile, line by line, to capture the vertical and horizontal coordinates of each CLLI code. First this function sets the file pointer to the second line of the file, skipping a header record. Then, this function proceeds to call fget() in order to read the bytes of data that contain the vertical and horizontal coordinates, from the nodefile, into the array node_info(). Upon return to the function, an end of line character is written to the array record, and the v and h coordinates are filtered out of the node_info() record, and into *vc, *hc.

This function begins by using the node index to calculate the node record's position in the ICF node file (an extra record must be skipped to account for a header record). Then, this function calls fget() in order to read node record from the nodefile into the buffer node_info[]. The node CLLI code field is then parsed and checked to make sure it is not a dummy code (all X's: "XXXXXXXXXXX"). If the CLLI code is valid, it is assigned to clli_ptr and the fields for the pointer s *vc and *hc (the vertical and horizontal coordinates) are assigned to be returned to the calling function

Inputs

character	linktype	The one-character ICF equipment code for a link/span record, denoting a narrowly defined set: `(T,D,E,N,G,L,C,W,Z,V,U,3,X,R,Y,I)`.

Outputs

returns	character	The converted TAMI equipment type code from the set: `(T1,L4,FO,MW)`.

Purpose This function is used to convert the ICF span/link type to a TAMI span type.

Called By `make_link()`

Calls To none

Local Variables

character	`returntype[]`	This output string holds the transformed input value for communication equipment code

Global Variables none

Algorithmic Description

This function is used to convert the ICF span/link type to a TAMI span type. The function is passed a one-character equipment code which it maps to a TAMI equipment code: T1, L4, FO (fiber optic), MW (microwave). See Appendix A for a description of ICF link type codes. The function returns the new type to the calling function.

3.4 array_make: Array Make Module

Purpose This module decodes and reformats span information from the node and link ICF data files; the span ("array") data is output to a file for use by the `mk_ncam_path` module.

Call Syntax `array_make`

Input Files

ICF node file	This file name is hard-coded to 'node_dat.icf.' See the ICF file description in Appendix A for more information regarding contents and format.
ICF link file	This file name is hard-coded to 'link_dat.icf.' See the ICF file description in Appendix A for more information regarding contents and format.

Output Files

array file	This file name is hard coded to 'array_make.out.' It is a binary file that contains a record number and pair of node index numbers for each span/link record stored in the `link_info[]` structure and reformatted from the ICF data files.
format	binary file, not viewable

Includes

`<stdio.h>`	Standard 'C' input/output functions
`"icf.h1"`	Defines constants used to process ICF files
`"fileio.c"`	User-defined I/O functions; see Appendix B

Constants Constants used in `array_make` are defined in `"icf.h1"` as follows:

`NODE_MAX 64`	number of characters in a ICF node file record
`Node_idx_fldlen 4`	number of characters in the node index field of the ICF node file
`LINK_MAX 24`	number of characters in a ICF link file record

Global Variables

file `*array_data_file`
pointer to the output array data file

integer

`link_num`	counts the number of links records placed into the `array_info[]` data structure
`count`	counts the number of records read from the ICF link file
`fp`	indicates the current file position when writing to output file
`i`	general loop count variable

structure `link_info[12000]` of type `link_array`
with fields:

integer	`link_info[].rec_no`	holds the span index number
character	`link_info[].link_pair`	holds both node index numbers of the span endpoints, treated as string

example

	rec_no	link_pair
link_info[i]	i	" 2591 97"
link_info[i+1]	i+1	" 2591 834"

Local Variables

Variables local to `main()`: none.

Component Functions

`make_array()`	loops through the node file to build the array data structure
`make_link()`	for a given node, makes an array record for each of the node's links given in the link file
`fget()`	reads in a given number of bytes from a given file position

Function Tree

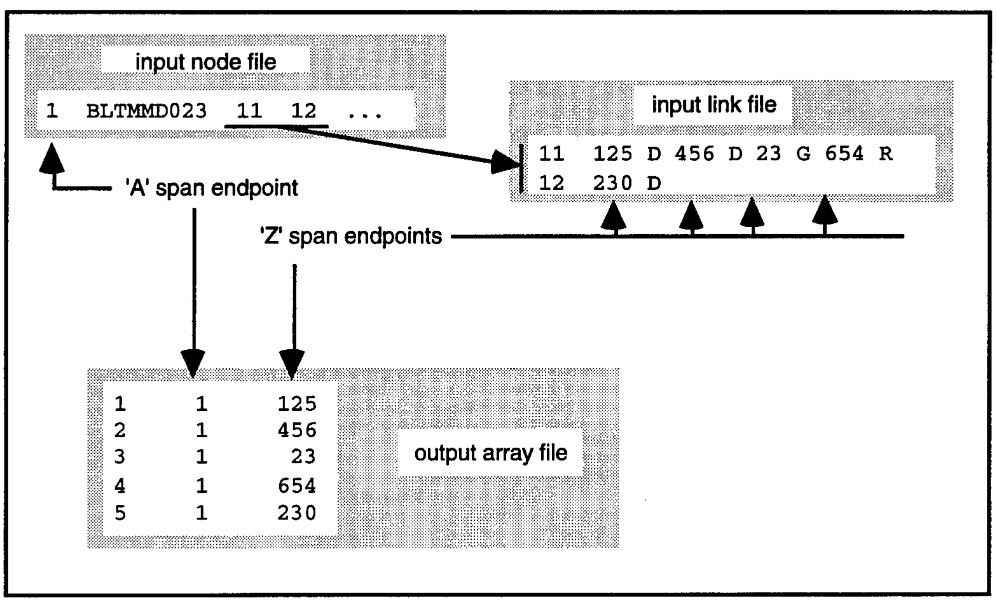
```
                             ┌─make_link()── fget()
main() ── make_array() ──────┤
                             └─ fget()
```

Algorithmic Description

The purpose of this module is to decode span information from the node and link ICF data files. Span/link records are the shortest identifiable segments of telecommunications transmission networks, typically representing segments between repeaters, or between a repeater and multiplexing/switching equipment. The `array_make` module outputs a list of these spans, designated by the node indices of the two span endpoints. This list is used in the `mk_ncam_path` module to represent long-haul switch-to-switch transmission routes as a series of component spans.

This module opens the hard-coded output file name, "`array_make.out`." It then calls `make_array()` (which in turn calls `make_link()`) to populate the `link_info[]` structure with spans. This structure is then written in binary form to the output file, the file is closed, and the total number of link/span records is written to the screen.

The following figure depicts the data flow performed by this module. For further description of ICF format, refer to Appendix A.



Inputs none; operates on global variables

Outputs

global `link_info[12000]` structure of type `link_array`

with fields:

integer	`link_info[].rec_no`	holds the span index number
character	`link_info[].link_pair`	holds both node index numbers of the span endpoints, treated as string

returns integer number of nodes in ICF node file for which links have been processed in the ICF link file

Purpose For each record in the ICF node file, this function builds the spans that terminate at that node from information contained in the ICF link file. These spans are stored in the `link_info[]` structure.

Called By `main()`

Calls To `make_link(), fget()`

Local Variables

file `*nodefile, *linkfile`
pointers to the input node and link ICF data files

integer	`link_head`	reference to the beginning of a node's ICF link file references
	`link_tail`	reference to the end of a node's ICF link file references
	`no_nodes`	the number of records in the ICF node file
	`idx_num`	general loop count variable
	`file_pos`	pointer to a particular byte or position in the ICF node file
	`status`	non-zero if the function's call to `make_link()` was successful
character	`clli_temp[]`	used to hold a node CLLI code
	`node_info[]`	used to hold a line/record from the ICF node file

Global Variables none

Algorithmic Description This function processes the ICF node file, record by record. For each node record, it parses the node CLLI code field and makes sure it is not a dummy code (all X's: `"XXXXXXXXXXX"`). Then it parses the `link_head` and `link_tail` fields from the node record. These fields point to the node's corresponding ICF link file records. If the `link_head` and `link_tail` fields are valid (non-zero), then `make_link()` is called, which actually steps through each of the node's link records and builds the spans. If the call to `make_link()` returns a `status` of `0`, then a general error message is printed to the output terminal. Although `make_array()` returns the number of records processed in the node file, this value is never used by the calling routine.

Inputs

integer	`node_idx`	the node file index of the current node being processed
	`head_pt`	the pointer to the first link file record describing the current node's links
	`tail_pt`	the pointer to the last link file record describing the current node's links
file	`*link_file`	the pointer to the link file

Outputs

global `link_info[12000]` structure of type `link_array`
with fields:

integer	`link_info[].rec_no`	holds the span index number
character	`link_info[].link_pair`	holds both node index numbers of the span endpoints, treated as string

integer `link_num` the number of records in `link_info[]`; also points to the next unused `link_info[]` record

returns integer a status indicator, equal to 1 if `make_link()` ran without error, or 0 if an error occurred in `make_link()`

Purpose Given a node index number and the starting and ending link records for that node, this function builds the spans that terminate at that node and places them in the `link_info[]` structure.

Called By `make_array()`

Calls To `fget()`

Local Variables

character	`linkrecord[LINK_MAX+1]`	used to read in and hold a record from the link file
	`linkpair[10]`	temporarily holds a pair of concatenated node index endpoints that define a link
long	`f_pos`	pointer to the current byte/position in the ICF link file
integer	`link_idx`	the index of a link endpoint in the link file
	`diff`	the number of link records to be processed for a given node, equal to the difference between `tail_pt` and `head_pt`
	`i`	refers to a position in `linkrecord[]` as it is sequentially processed
	`j`	counts the number of link records processed, up to `diff`
	`status`	equal to 1 if `make_link()` ran without error, 0 if an error occurred

Global Variables see Outputs above.

Algorithmic Description In the ICF file format, a node file record points to the range of link file records that describes the other endpoint of links that use that node. Each link file record has fields reserved for up to four such link endpoints. For a given node, starting link record, and ending link record, function `make_link()` builds each link/span and stores it in the `link_info[]` structure. If an error occurs, `make_link()` returns a status of 0.

The function starts by calculating diff, the number of link records to process from head_pt to tail_pt. It then enters a loop to read each link record from head_pt to tail_pt into linkrecord[], and to verify that no file read error occurred. For each link record, the function scans link endpoint fields until all four link endpoints have been processed or a blank field is encountered. Processing of each link endpoint, stored in link_idx, entails the following:

- Concatenating the span endpoints, (node_idx, link_idx) into linkpair
- Assigning the current span/link index (link_num) to the rec_no field of the span/link record, link_info[link_num].rec_no
- Copying linkpair into the link_pair field of the span/link record, link_info[link_num].link_pair
- Incrementing the span/link counter, link_num
- Moving i to point to the next link field in the linkrecord[] (points to the end of the string if done)

After each link field of each link record has been processed, and the resulting spans stored in link_info[], make_link() returns the value status to the calling function.

3.5 mk_ncam_path: Make Paths Module

Purpose

This module decodes and reformats path information from the ICF data files and the output array file from the `array_make` module; whereas the ICF files encode physical transmission paths as a series of indexed nodes, TAMI must use paths that are encoded as a series of spans, where each span has a characteristic span type that can be damaged. `Mk_ncam_path` employs code and data types from the IDA/CAM code library. Readers are directed to Reference 4, the IDA/CAM Programmer's Manual where appropriate.

Call Syntax

`mk_ncam_path`

Input Files

<u>ICF node file</u>	This file name is hard-coded to 'node_dat.icf.' See the ICF file description in Appendix A for more information regarding contents and format.
<u>ICF link file</u>	This file name is hard-coded to 'link_dat.icf.' See the ICF file description in Appendix A for more information regarding contents and format.
<u>ICF trunk file</u>	This file name is hard-coded to 'cg_dat.icf.' See the ICF file description in Appendix A for more information regarding contents and format.
<u>ICF path file</u>	This file name is hard-coded to 'pid_dat.icf.' See the ICF file description in Appendix A for more information regarding contents and format.
<u>array file</u>	This file name is hard coded to 'array_make.out.' It is a binary file that contains a record number and pair of node index numbers for each span/link record stored in the `link_info[]` structure and reformatted from the ICF data files.
format	binary file, not viewable

Output Files

<u>path file</u>	This file name is hard coded to 'mk_ncam_path.out.' Each record in this file specifies a physical transmission path between a pair of switches. A physical transmission path is defined by the series of spans (from none for collocated switches to a maximum of 662) that connect a switch pair. Spans are identified by indexes that point to the appropriate record number in the span file.
format	<switch CLLI A>, <switch CLLI Z>, <span1>, <span2>, ... , <span n> (c11, 1x, c11, 1x, i6, i6, ... , i6)
example	`ADMSTX0101T AKRNOH2505T    3045   3042   7579    237` (where `'3045 . . . .'` are record numbers in the span file output by the `span_make` module)

Includes

`"nat.h"` IDA/CA M include file (Reference 4)

Constants

None

Global Variables

cg_path	`*first_cg`	points to the first cg in the cg structure defined in Reference 4; uses user-defined type
integer	`count`	counts the number of records in the array input file
	`first_flag`	indicates whether the first path of a trunk group is being processed
	`err`	toggle to indicate error in command line arguments

structure `link_info[12000]` of type `link_array`
with fields:

integer	`link_info[].rec_no`	holds the span index number
string	`link_info[].link_pair`	holds both node index numbers of the span endpoints, treated as string

Local Variables

Variables local to `main()`:

character	`c`	a single character used to parse command line options
	external `*optarg`	points to the command line argument string supplied externally by the operating system
cg_path	`*cg_node`	points to a record in the cg structure defined in Reference 4; uses user-defined type
int	`i`	general loop count variable
	external `*optind`	points to the number of command line arguments processed, supplied externally by the operating system
file	`*array_data_file, *outfile`	pointers to the input array file and output path file

Component Functions

`write_span()`	writes out the path, including all the spans, for a given circuit group record
`bsearch()`	standard 'C' binary search routine, used to search for spans in the `link_info[]` structure
`qsort()`	standard 'C' sort routine, used to sort `link_info[]`
`span_cmp()`	string comparison function used by `bsearch()`; see `char_comp()` in Appendix B for description
`sort_cmp()`	string comparison function used by `qsort()`; see `char_comp()` in Appendix B for description
`get_data()`	this IDA/CAM routine loads all four ICF files into the global IDA/CAM data structures defined in "`nat.h.`" See Reference 4 for a description of this function

Function Tree

```
          ┌ get_data()
main() ───┼ qsort() ── sort_cmp()
          └ write_span() ── bsearch() ── span_cmp()
```

Algorithmic Description

As described in Appendix A, the ICF paths are encoded as a series of network nodes. TAMI requires that paths be described as series of links/spans. This module uses the list of spans created by the `array_make` module to convert ICF paths to TAMI paths and replaces indexed nodes with their CLLI codes.

The module starts by loading the input files into structures. First, it opens the file `array_make.out` and loads its records into the `link_info[]` structure. This structure is then sorted by `qsort()`. The IDA/CAM routine, `get_data()` is called to load the four ICF files into data structures.

The main algorithm is then performed using the linked list of circuit group data structures. Each of these structures contains a pointer to the linked list of all the paths used by the circuit group. Each path structure points to the linked list of all the nodes that form the path. All of these structures are defined in more detail in Reference 4. `Mk_ncam_path` steps through the linked list of circuit groups, calling `write_span()` to build and output the circuit group's paths in TAMI format.

Inputs

file	`*file`	points to the output path file
cg_path	`*node`	points to the circuit group structure for which paths are to be built; cg_path is a user-defined structure described in Reference 4

Outputs none; prints results directly to output file

Purpose For each circuit group record in the ICF cg file, this function uses TAMI spans stored in the `link_info[]` structure to build the paths used by the circuit group.

Called By `main()`

Calls To `bsearch()`

Local Variables

pid_tbl	`*node_pid`	pointer to a table of a paths for a given circuit group; pid_tbl type defined in Reference 4
path_node_tbl	`*npath`	pointer to a particular path; path_node_tbl type defined in Reference 4
link_array	`*span`	pointer to an element of the `link_info[]` structure
integer	`index1, index 2`	the ICF node indexes of span endpoints
	`cnt`	counts the number of paths processed for a circuit group
	`cnt1`	counts the number of spans processed for a circuit group
character	`buf[]`	holds a span record in the same format as `link_info[].link_pair`

Global Variables

integer	`first_flag`	indicates whether the first path of a trunk group is being processed

Contains global IDA/CAM structures declared in `"nat.h"` and loaded by `get_data()`; see Reference 4

Algorithmic Description

For each circuit group record in the ICF cg file, this function uses TAMI spans stored in the `link_info[]` structure to build the paths used by the circuit group. Specifically, `write_span()` performs the following steps:

1) Uses the pointer to the current circuit group, `*node`, to access the path endpoint CLLI codes and a pointer to the cg's table of paths

2) Loops through each path in the path table

3) For each path, loops through all the nodes that compose the path, searching `link_info[]` to convert each chained pair of nodes into a link/span index

The function prints its results to the output file as they are obtained.

3.6 mat_trk: Match Trunks module

Purpose

For every pair of IEC backbone switches, this program maps the logical trunk group to the corresponding physical transmission paths, as described in Algorithmic Description below. Bi-directional trunk groups are the default, but one-way trunk groups are assumed for MCI.

Call Syntax

`mat_trk -f <filename> [options]`

options:

`-a`	turn off substitution of average trunk size
`-d`	turn debug mode on to print debug statements
`-f`	reads in input file `<filename>` which contains a list of the 5 input/output files used by `mat_trk`
`-m`	specify MCI data, which expects both 4 character switch CLLI code and a 3 character location code in the switch input file
`-o`	assume trunk groups are one-way (e.g., for MCI)
`-?`	user help--prints call syntax and exits without running

example `mat_trk -f MCIfiles.fy94 -o -m -a` (spaces optional)

Input Files

list file

This file simply contains the names of the three input files and two output files to be used by `mat_trk`. File names are limited by `mat_trk` to a length of 50 characters.

format

line1:	<path file name>
line2:	<switch file name>
line3:	<trunk file name>
line4:	<output trunk file name>
line5:	<output path file name>

switch file

This file contains the list of IEC backbone switches. For MCI data, this file also contains a location code for each switch. Non-MCI switches are specified by an 11-character CLLI code, where the first 8 characters identify a unique location. MCI switches are specified by a 4 character code, and locations are given by a separate 3 character code.

format:

Non-MCI:	<IEC switch CLLI code> (c11)
MCI:	<MCI switch code>, <MCI location code> (c4, 1x, c3, 3x)

input trunk file

This file, created by the `cg_make` module, specifies the trunk groups and quantities for the IEC backbone, specified by the end point CLLI codes and an integer number of trunks. If trunk groups are assumed to be one-way, then the first CLLI code is the originating switch.

format:

<switch CLLI A>, <switch CLLI Z>, <trunk quantity>
(2x, c11, c11, i4)

path file

Each record in this file, created by the `mk_ncam_path` module, specifies a physical transmission path between a pair of switches. A physical transmission path is defined by the series of spans (from none for collocated switches to a maximum of 662) that connect a switch pair. Spans are identified by indexes that point to the appropriate record number in the span file.

format: <switch CLLI A>, <switch CLLI Z>, <span1>, <span2>, ... , <span n>
(c11, 1x, c11, 1x, i6, i6, ... , i6)

Output Files

<u>output trunk file</u> The output trunk file specifies the IEC switch CLLI codes of the trunk endpoints, the number of trunks in the A to Z direction (or all bi-directional trunks), and the number of trunks in the Z to A direction (only used for one-way trunk groups). Each record is in 1-to-1 correspondence with the output path file. That is, the number of trunks in the n^{th} trunk file record traverse the transmission path specified by the n^{th} path file record. Therefore, when a trunk group is split over 5 paths for example, there will be 5 trunk file records to describe this, corresponding to each of the 5 path file records.

format: <switch CLLI A>, <switch CLLI Z>, <A->Z trunk quantity>, <Z->A trunk quantity>
(c11, 1x, c11, 1x, i4, 1x, i4)

<u>output path file</u> The output path file specifies the paths used to implement the trunk groups. If a path has no corresponding trunk group record, then the average trunk group size may be used, or the path record can be thrown out. There are two differences from the input path file: (1) if the path had invalid or non-switch endpoints it has been filtered out; (2) paths that could not be matched to a corresponding trunk group have been filtered out of the output file if the `[-a]` option was used.

format: same as input path file

Includes

`<stdio.h>`
`<stdlib.h>`
`<string.h>`
`"fileio.c"` See Appendix B

Constants

`PATH_REC`	4000	maximum number of characters in a path file record
`CLLI_LNG`	12	length of a switch CLLI code, including terminating null character
`SWLOC_LNG`	4	length of an MCI location code, including terminating null character
`SWITCH_MAX`	200	maximum number of records in the switch file
`TRK_MAX`	15000	maximum number of records in the trunk file

Global Variables

file: `*pathfile, *switchfile, *trunkfile, *filelist, *outfile, *outfile2` pointers to the input and output files

integer:

`tog_mci`	toggle to indicate `-m` command line option is being used
`tog_spr`	toggle for a Sprint option that is no longer used
`tog_debug`	toggle to indicate `-d` command line option is being used
`tog_noavg`	toggle to indicate `-a` command line option is being used
`numswitch`	the number of records read in from the switch file
`numtrunk`	the number of validated records read in from the trunk file
`maxnpath`	indicates the maximum number of path file records that pertain to a single switch pair

float	`avgtrunk`	the average size of the valid trunk records
character	`switches[SWITCH_MAX][CLLI_LNG]`	array to hold up to `SWITCH_MAX` IEC switch CLLI codes, each of length `CLLI_LNG` (including null character)
	`loc[SWITCH_MAX][SWLOC_LNG]`	array to hold up to `SWITCH_MAX` IEC switch location codes (used only for MCI data), each of length `SWLOC_LNG` (including null character)
	`maxAsw[CLLI_LNG], maxZsw[CLLI_LNG]`	these hold the CLLI codes of the switch pair with the maximum number of paths, tracked as a statistic printed to the screen
	`buffer[100][PATH_REC]`	this buffer holds up to 100 path records to support the sequential processing of the path file

structure `trks[TRK_MAX]` of type `trk_struct`
with fields:

character	`trks[].clliA`	originating trunk group endpoint
	`trks[].clliZ`	terminating trunk group endpoint
integer	`trks[].qty`	quantity of trunks in trunk group record
	`trks[].used`	toggle to track use of trunk group record

Component Functions

`openfiles()`	opens input and output files
`loadswitches()`	loads switch file into the switch list, `switches[]`
`loadtrunks()`	loads trunk file records into the trunk structure
`outprint()`	prints out one output trunk file record for each path
`qsort()`	standard 'C' quick sort routine, used to sort switch list
`bsearch()`	standard 'C' binary search routine, used to search switch list
`char_comp()`	string comparison routine for `qsort()` and `bsearch()`
`processpaths()`	maps a logical trunk group to its physical paths
`getsize_oneway()`	finds the oneway trunk sizes given a pair of switch endpoints
`getsize()`	finds the bi-directional trunk size given a pair of switch endpoints
`getswidx()`	finds the index number of a switch within the switch list
`processtrunks()`	handles trunk groups which had no matching paths

Function Tree

```
          ┌─openfiles()
          ├─loadswitches() ── qsort() ── char_comp()
          ├─loadtrunks() ──── bsearch() ── char_comp()
main() ───┼─processpaths() ─┬─ bsearch() ── char_comp()
          │                 ├─ getsize_oneway()
          │                 ├─ getsize()
          │                 └─ outprint()
          └─processtrunks() ┬─ getswidx()
                            ├─ getsize_oneway()
                            └─ outprint()
```

Algorithmic Description

For every pair of IEC backbone switches, this module maps the logical trunk group to the corresponding physical transmission paths. Because the trunk group and path information are often maintained by separate units within a carrier's organization, there is no guarantee that the logical trunk groups map cleanly to each physical path. This module addresses this problem by enforcing a number of "set recombination" rules:

1) *Invalid endpoints.* Trunk groups and paths that do not have switch endpoints (identified in the switch file) are filtered out and not used

2) *Paths and trunk groups match.* Where paths can be matched to a corresponding trunk group (i.e., path endpoints are the same as trunk group endpoints), the module divides the trunks in the trunk group equally among all available paths (with any integer remainder of trunks given to the last path). This is the most reasonable assumption in the absence of more specific carrier data. For example, a trunk group of size 60 that has 5 matching physical paths would be divided into 5 output trunk group records, each of size 12.

3) *Path, but no trunk group.* Where a path cannot be matched to a corresponding trunk group, the module gives two options. One, throw the path out. Or two, make up a trunk group containing the average number of trunks in the network, based on the reasoning that a trunk group should exist for the path, but was inadvertently left out of the trunk group data. The first option is usually employed.

4) *Trunk group, but no path.* Where a trunk group exists with no corresponding physical path data, the program checks to see if the endpoints are collocated. If so, no path is necessary since the switches are in the same building, and the trunk group size can be used. A path record is generated (containing no spans) to maintain the one-to-one mapping between trunk groups and paths. If the switches are not collocated, the trunk record must be thrown out since there is no corresponding path data with which to evaluate network damage.

The module provides the option `[-o]` to model trunk groups as one-way, although paths are always considered to be bi-directional. In the case of one-way trunk groups, the module maintains separate A–>Z and Z–>A trunk size fields for each switch pair. One-way trunk groups are used for MCI data. The module also provides an MCI option `[-m]` to indicate that the switch file will contain 4 character switch codes and 3 character location codes, and not the usual 11 character CLLI codes. The `[-a]` option tells the module not to assume an average trunk size for paths that have no corresponding trunk groups.

After the command line arguments have been parsed and interpreted, `main()` executes in the following order:

1) Calls `openfiles()` to open pointers to input and output files

2) Calls `loadswitches()` to read the switch file into `switches[]`

3) Calls `loadtrunks()` to load the trunk file into the trunk structure, `trks[]`. Trunk records with non-switch end points are thrown out

4) Calls `processpaths()` to sequentially process the sorted path file and find matching trunk groups. The `trks[].used` field is set for trunk groups that are matched here. Output paths and trunks are written to the output files as they are matched

5) Calls `processtrunks()` to handle all of the trunks that didn't have matching paths. If the endpoints are collocated, it uses the trunk group; otherwise, it must be thrown out

6) Prints out the switch pair that had the greatest number of physical paths, then exits.

Inputs

character	`files`	string containing the name of the file that lists six input/output files

Outputs

global	file	`*filelist`	points to the file whose name is contained in the `files` string
		`*pathfile`	points to a file that contains information on the physical paths and switches
		`*switchfile`	points to a file that contains information on the backbone network switches
		`*trunkfile`	points to a file that contains information on the logical trunks
		`*outfile`	filtered version of trunkfile
		`*outfile2`	filtered version of pathfile

returns no formal values are returned

Purpose To open path, trunk and switch input files and path and trunk output files.

Called By `main()`

Calls To none

Local Variables

character	`tempfile[50]`	this variable is used temporarily to hold the name of the next file to be opened and read in from the list of files in `filelist`

Global Variables none

Algorithmic Description

This function opens the file whose name is stored in the string `files`, setting a filepointer to `filelist. Filelist` contains a list of all the input files to be opened in the following order: `pathfile, switchfile, trunkfile, outfile and outfile2` This function then proceeds to open each of these files, in the order they are read in from `filelist`, assigning them to the matching filepointers.

Errors encountered during any file opening operation result in an error message being printed to the screen, and termination of the module.

Inputs	none; operates on global variables		
Outputs			
global	character	`switches[]`	the list of IEC switch CLLI codes
		`loc[]`	the list of MCI switch locations
	integer	`numswitch`	used to count the number of records in the switchfile
	file	`*switchfile`	points to the switch file
returns	no formal values are returned		

Purpose To read from the switch file, to load the switch CLLI codes into the `switches` vector, to count the number of switch records, and to perform an alphabetical sort

Called By `main()`

Calls To `qsort()` the quick sort routine from the standard 'C' library `stdlib.h`

Local Variables

integer	`k`	general loop count variable
	`j`	not used
character	`line[]`	used to hold a line of input from the switch file
	`tempCLLI[]`	temporarily holds a parsed switch name
	`temploc[]`	temporarily holds a parsed MCI switch location
	`tempsw[]`	temporarily holds a parsed MCI switch CLLI code

Global Variables

integer	`tog_mci`	a toggle used to indicate that MCI switches and locations should be expected in switch file

Algorithmic Description

This function reads the switch file line by line, loading each CLLI code into the array of strings `switches[]`, and setting `numswitch` to the total number of switches in the switch file. Because other functions in this module depend on the switches being in alphabetical order, the routine passes the `switches[]` array to `qsort()`.

If MCI data is indicated by `tog_mci`, then each switch name contains a four character switch CLLI code and a three character location code. In this case the function loads the CLLI codes into `switches`, and the location codes into `loc[]`.

Inputs none; operates on global variables

Outputs

global	trk_struct structure *with fields*	trks[]	used to hold each record in the trunk file
	character	trks[].clliA	originating trunk group endpoint
		trks[].clliZ	terminating trunk group endpoint
	integer	trks[].qty	quantity of trunks in trunk group record
		trks[].used	toggle to track use of trunk group record

returns no formal values are returned

Purpose To read the trunk file, loading valid records into the trunk structure as described in the algorithmic description below

Called By `main()`

Calls To

`char_comp()`	the function used to specify ascending alphabetic order in the character comparison performed by `bsearch()`
`bsearch()`	the binary search routine from the standard 'C' library `stdlib.h`

Local Variables

integer	`k, j`	general loop count variables
	`*found_A`	integer pointer used to indicate whether the originating end point of a trunk group is a valid switch: zero if false and non-zero if true
	`*found_Z`	integer pointer used to indicate whether the terminating end point of a trunk group is a valid switch: zero if false and non-zero if true
	`tottrunks`	total number of trunks for the IEC that have valid switch end points

Global Variables

integer	`numtrunk`	the number of valid trunk group records read from the trunk file
float	`avgtrunk`	the average size of a valid trunk group: `tottrunks/numtrunk`

Algorithmic Description

This function handles cases where a trunk group record did not have a corresponding physical path . If the trunk group endpoints are collocated, then a physical path is unnecessary. The trunk group is used (written to the output trunk file) and a corresponding dummy path record (path with no span) is written to the output path file to maintain the required one-to-one mapping between logical trunk groups and physical paths. Trunk groups that do not have collocated endpoints are filtered out of the data and are not used. Because these trunk groups do not have physical path information, it is impossible to evaluate the effect of network damage on them. The function logic is as follows.

This function reads the trunk file line by line, until the end of the file is reached. For each line (record), it parses the originating trunk end point, terminating trunk end point, and trunk quantity, and loads these fields into the `trks[]` structure. Next, the function conducts a binary search to check that the trunk end points, `trks[].clliA` and `trks[].clliZ`, are found in the list of switches, `switches[]`. If so, the trunk record is valid, the loop counter is advanced, and the trunk quantity is summed into `tottrunks`. If not, the trunk record is not valid and the loop counter is not advanced, so that the next record read in from the trunkfile will overwrite it.

When the end of the file is reached, `numtrunk` is set to the loop counter, and specifies the number of valid records read. `Avgtrunk` is computed as `tottrunks/numtrunk`, and both `numtrunk` and `avgtrunk` are printed to the standard output.

Inputs

integer	tog_oneway	a command line toggle used to indicate the use of one way trunks

Outputs

global	character	buffer[]	a vector of strings that holds valid paths before printing them to an outfile
		maxAsw[]	used to hold originating endpoint
		maxZsw[]	used to hold terminating endpoint

returns no formal values are returned

Purpose To process paths one by one and write out a filtered path file and filtered trunk file in one-to-one correspondence with the paths as described in the algorithmic description below

Called By main()

Calls To

char_comp()	the function used to specify ascending alphabetic order in the character comparison performed by bsearch()
bsearch()	the binary search routine from the standard 'C' library stdlib.h
getsize_oneway()	the subroutine used to determine the size for oneway trunks.
outprint()	This routine is used to split trunks among paths and print results to an outfile
getsize()	the subroutine used to determine the size for bi-directional trunks.

Local Variables

integer	k	general loop count variable
	pathrec	counts records read in from pathfile
	pathctr	counts number of paths for a given switch pair
	newpath	toggle to detect next switch pair
	avgused	toggle to detect use of average trunk size constant avgtrunk
	qtyA	trunk size variable for A->Z direction
	qtyZ	trunk size variable for Z->A direction
	*found_A	integer pointer used to indicate whether the originating end point of a trunk group is a valid switch: zero if false and non-zero if true
	*found_Z	integer pointer used to indicate whether the terminating end point of a trunk group is a valid switch: zero if false and non-zero if true
character	pathline	a temporary string variable to hold a record from the pathfile
	pathA	holds the originating endpoint of the path in pathline
	pathZ	holds the terminating endpoint of the path in pathline
	nextA	holds the originating endpoint of 'next' path to compare it with current path
	nextZ	holds the terminating endpoint of 'next' path to compare it with current path

Global Variables

float	`avgtrunk`	the average size of a valid trunk group: `tottrunks/numtrunk`

Algorithmic Description

This function reads the first record of the path file, increments the path record counter, `pathrec`, parses the originating and terminating endpoints, and loads these into `pathA` and `pathZ`, respectively. Next the function conducts a binary search to check that the path endpoints are found in the list of switches, `switches[]`. If so, the path record is valid, the path record is copied to `buffer`, and the loop counter is advanced.

Next this function reads the remaining records in the path file line by line. For each record it parses the originating and terminating endpoints and loads them into `nextA` and `nextZ` respectively, then a comparison is made between this pair of endpoints and `pathA, pathZ`. If the endpoints are a match, this record is copied to `buffer`, and the next record is processed in the same manner. (If there is not a match, these values are held in `nextA, nextZ` for the next iteration of this section.) When all sets of matching endpoints have been found, the function calls a subroutine, (`getsize_oneway or getsize`), which return the trunk size for the `pathA, pathZ` switch pair. Then, this trunk size is used in a call to the `outprint` subroutine, to divide the trunk size evenly among the matching sets of switch pairs held in the buffer, with any remainder going to the last switch pair.

Finally, this function sets `nextA, nextZ`, the last read in switch pair, (which were also the first non-matching pair) equal to `pathA, pathZ`, and loops back to searching the path file for matching pairs.

Inputs			
integer	`tog_oneway`	toggle used to indicate one-way trunk groups for MCI	
Outputs			
global	trk_struct structure	`trks[]`	used to hold each record in the trunk file
	with fields		
	character	`trks[].clliA`	originating trunk group endpoint
		`trks[].clliZ`	terminating trunk group endpoint
	integer	`trks[].qty`	quantity of trunks in trunk group record
		`trks[].used`	toggle to track use of trunk group record
	string	`loc[]`	an array used to determine collocation for MCI
returns	no formal values are returned		

Purpose To handle cases where a trunk group exists with no corresponding path record.

Called By `main()`

Calls To

`outprint()`	To split trunks and print results to an outfile
`getsize_oneway()`	To add up the trunk quantity for oneway trunk group
`getswidx()`	To return the index of a switch CLLI within the switch list.

Local Variables

integer	`i`	general loop count variable
	`qtyZ`	trunk size variable for Z->A direction
	`idxA`	A value returned by getswidx, the index of the trunk group originating endpoint
	`idxZ`	A value returned by getswidx, the index of the trunk group terminating endpoint
	`tot_unused`	used to keep track of the total number of trunk records that were not used.

Global Variables

integer	`numtrunk`	the number of valid trunk group records read from the trunk file
	`tog_mci`	a toggle used to indicate MCI trunks

Algorithmic Description

For each trunk record, this function checks to see if the record has been used. If it has not then it checks for MCI trunk record (`tog_mci`). If the records are not MCI then it uses a matching process to determine the collocation, and to create a dummy path with no spans and a corresponding trunk size record is written to the output file. If the trunk record is MCI then the function uses `loc[]` to determine collocation.

Finally, this function prints to screen all trunk records that did not get used, as well as a total.

Inputs			
character	`*head`	a character pointer used to represent the originating end point of a trunk group	
	`*tail`	a character pointer used to represent the terminating end point of a trunk group	
Outputs			
global	trk_struct structure *with fields*	`trks[]`	used to hold each record in the trunk file
	character	`trks[].clliA`	originating trunk group endpoint
		`trks[].clliZ`	terminating trunk group endpoint
	integer	`trks[].qty`	quantity of trunks in trunk group record
		`trks[].used`	toggle to track use of trunk group record
returns	integer	returns the oneway trunk size for the switch endpoints passed in	
Purpose	To process the trunk size for oneway trunks.		
Called By	`processpaths()` `processtrunks()`		
Calls To	none		
Local Variables			
integer	`i`	general loop count variable	
Global Variables			
integer	`numtrunk`	an integer holding the number of trunks	
Algorithmic Description	This function loops through the trunk group list, summing all trunk quantities with end points `(head, tail)`. This total is returned by `getsize_oneway()`. The `trks[].used` field is set for trunks that are used.		

Inputs

character	`*head`	a character pointer used to represent the originating end point of a trunk group	
	`*tail`	a character pointer used to represent the terminating end point of a trunk group	

Outputs

global	trk_struct structure	`trks[]`	used to hold each record in the trunk file
	with fields		
	character	`trks[].clliA`	originating trunk group endpoint
		`trks[].clliZ`	terminating trunk group endpoint
	integer	`trks[].qty`	quantity of trunks in trunk group record
		`trks[].used`	toggle to track use of trunk group record
returns	integer	returns the bi-directional trunk size for the switch endpoints passed in	

Purpose To compute and return the total number of trunks for a given switch pair.

Called By `processpaths()`

Calls To none

Local Variables

integer	`i`	general loop count variable
	`tot`	used to hold the total number of trunks to be returned

Global Variables none

Algorithmic Description

This function loops through the trunk group list, summing all trunk quantities with end points `(head, tail)` or `(tail, head)`. This total is returned by `getsize()`. The `trks[].used` field is set for trunks that are used. If no trunks are found, (-1) is returned.

Inputs			
character	*swclli	the CLLI code of the switch for which to search	
Outputs			
global	character	switches[]	The list of IEC switch CLLI codes to search
returns	integer	the index of swclli in vector switches[], -1 if not found	

Purpose This short function returns swclli's index within switches[] or -1 if it is not a member of the list

Called By processtrunks()

Calls To none

Local Variables
integer i general loop count variable

Global Variables none

Algorithmic Description This function uses a loop to sequentially compare swclli to each element in switches[]. It returns the array index of the first (and only) element that matches, or -1 if no match is found..

Inputs			
character	`*head`	a character pointer used to represent the originating end point of a trunk group	
	`*tail`	a character pointer used to represent the terminating end point of a trunk group	
	`qtyA`	trunk size variable for A->Z direction	
	`qtyZ`	trunk size variable for Z->A direction	
	`pathctr`	counts number of paths for a given switch pair	
Outputs			
global	file	`*outfile`	filtered version of the trunk file
returns	no formal values are returned		
Purpose	To split trunks across paths and print results to an outfile		
Called By	`processpaths()` `processtrunks()`		
Calls To	none		
Local Variables			
integer	`i`	general loop count variable	
Global Variables	none		
Algorithmic Description	This function calculates the ratio of trunk size (`qtyA, qtyZ`) to the number of paths (`pathctr`) for each (`head, tail`) switch pair and prints the results to `outfile`.		

3.7 rem_dups: Remove Duplicate Records Module

Purpose

This module addresses an artifact of MCI data, namely duplicate records in the path data file. This module removes path records that are within two records "distance" of a duplicate record

Call Syntax

`rem_dups -i <input file> -o <output file>` *[options]*

mandatory:

`-i <input file>`	specifies the name of the file containing the input path file records
`-o <output file>`	specifies the name of the file which will hold the filtered output path file

options:

`-?`	user help–prints call syntax and exits without running

example

```
rem_dups -i path_file.MCI -o rem_dup.out
```

Input Files

<u>input path file</u>

Each record in this file, specifies the physical transmission path between a pair of switches. A physical transmission path is defined by the series of spans (from none for collocated switches to a limit of 662) that connect a switch pair. Spans are identified by indexes that point to the appropriate record number in the span file.

format <switch CLLI A>, <switch CLLI Z>, <span1>, <span2>, ... , <span n>
(c11, 1x, c11, 1x, i6, i6, ... , i6)

example

```
AST1     NYC1     3045  3042  7579  237
NYC1     AST1     3045  3042  7579  237
NYC1     AST2     3045  3042  7579  237
NYC1     AST2     3045  3042  7579  237
```

Output Files

<u>output path file</u>

This file is in the same format as the input path file, except that duplicate records have been removed, and switch endpoints have been arranged to ensure that the first switch is alphabetically prior to the second switch.

format same as input path file

example

```
AST1     NYC1     3045  3042  7579  237
AST2     NYC1     3045  3042  7579  237
```

Includes

`<stdio.h>`
`<stdlib.h>`
`<string.h>`
`"fileio.c"` see Appendix B

Constants

`PATH_REC 4000` maximum number of characters in a path file record

Global Variables

none

Local Variables

Variables local to `main()`:

extern	character	`*optarg`	points to a command line argument.
	integer	`optind`	not used
file	`*infp, *outfp`	pointers to the input and output files	
integer	`tog_infile`	toggle to indicate input "file name" was read in successfully	
	`tog_outfile`	toggle to indicate output "file name" was read in successfully	
	`tog_err`	toggle to indicate an error in the command line arguments	
	`line2_init`	signals that `line2` has been initialized with a string value	
	`max_path_len`	keeps track of the maximum path record length	
	`len1`	holds the length of the path record in `line1`	
	`lennext`	holds the length of the path record in `nextline`	
	`i`	general loop count variable	
character	`ch`	holds the command line argument	
	`infile[]`	holds the name of the input file	
	`outfile[]`	holds the name of the output file	
	`sw1[]`	holds a path's originating endpoint	
	`sw2[]`	holds a path's terminating endpoint	
	`line1[PATHREC]`	holds a line read in from the input path file	
	`line2[PATHREC]`	holds a line read in from the input path file	
	`nextline[PATHREC]`	holds a line read in from the input path file	

Component Functions

none

Function Tree

none, `main()` only

Algorithmic Description

This module addresses an artifact of MCI data, namely duplicate records in the path data file. This module removes path records that are within two records "distance" of a duplicate record

This module consists solely of a `main()` routine . The `main()` routine first conducts initialization steps: defines local variables; processes command line arguments; reads in the name of the input file and opens it in read only mode; and reads in the name of the output file and opens it in write only mode.

The main algorithm of `rem_dup()` tracks the last two unique path file records, with which subsequent records are compared to determine duplication. The main algorithm is initialized by loading the first path record into `line1` and printing this line to the output file. Subsequent records are read into `nextline` and compared to `line1`. As soon as a second unique path record is found, it is copied from `nextline` to `line2` and written to the output file..

Once `line1` and `line2` are initialized , they are used as a queue data structure to hold the last two unique path records. When `nextline` is found to be different from `line1` and `line2`:

- `nextline` is printed to the output file
- `line1` shifted out of queue
- `line2` is shifted to `line1`
- `nextline` is shifted to `line2`

Pathfile records are always printed out such that the endpoint CLLI codes are in alphabetical order. The algorithm above is continued until the end of the file is reached.

3.8 sort_paths: Sorting of path file module

Purpose

This module is used to sort the records in the MCI path file

Call Syntax

sort_paths -i <input file> -o <output file> *[options]*

mandatory:

-i <input file> specifies the name of the file containing the input path file records

-o <output file> specifies the name of the file which will hold the filtered output path file

options:

-? user help–prints call syntax and exits without running

example sort_paths -i infile -o outfile

Input Files

<u>input path file</u>

Each record in this file specifies the physical transmission path between a pair of switches. A physical transmission path is defined by the series of spans (from none for collocated switches to a limit of 662) that connect a switch pair. Spans are identified by indexes that point to the appropriate record number in the span file.

format <switch CLLI A>, <switch CLLI Z>, <span1>, <span2>, ... , <span n>
(c11, 1x, c11, 1x, i6, i6, ... , i6)

example

AST1	NYC2	3045	3042	7579	237
AST1	NYC1	3045	3042	7579	237

Output Files

<u>output path file</u>

This file is in the same format as the input path file, except that records have been sorted in alphabetical ascending order.

format same as input path file

example

AST1	NYC1	3045	3042	7579	237
AST1	NYC2	3045	3042	7579	237

Includes

<stdio.h>
<stdlib.h>
<string.h>
"fileio.c" see Appendix B.

Constants

PATH_REC	3700	maximum number of characters in a path file record
NUM_PATH	12700	maximum number of path records capable of being read

Global Variables

integer num_rec counts the number of path records read from the input file

character paths[][] an array used to store every record read from the input file for use during the sorting routine .

file *infp, *outfp pointers to the input and output files

Local Variables Variables local to `main()`:

extern	character	`*optarg`	points to the current command line argument being parsed
	integer	`optind`	not used
integer	`tog_infile`	toggle to indicate input file was read in successfully	
	`tog_outfile`	toggle to indicate output file was read in successfully	
	`tog_err`	toggle to indicate an error in the command line arguments	
	`i`	general loop count variable	
character	`ch`	used to parse command line options	
	`infile[]`	holds the name of the input file	
	`outfile[]`	holds the name of the output file	

Component Functions

`char_comp()`	string comparison routine for `qsort()`.
`qsort()`	standard 'C' sorting routine

Function Tree

```
main() —qsort() —char_comp()
```

Algorithmic Description

This module is used to sort the records in the path file for MCI data. Essentially it reads the input file into an array, passes the array to the standard 'C' sort routine and prints the result to the output.

The `main()` routine first conducts initialization steps: defines local variables; processes command line arguments ; reads in the name of the input file and opens it in read only mode; and reads in the name of the output file and opens it in write only mode.

Then, the main algorithm of this function loads each record in the input file into the array `paths[][]` and passes the array to the standard 'C' `qsort()` routine. `Qsort()` uses `char_comp()` to sort records in ascending order and modifies the records of `paths[][]` until they are completely sorted. Finally this module prints the sorted records from `paths[][]` to the output file

3.9 clli3_4: Location to Switch Code Conversion module

Purpose This module addresses a shortfall in MCI data only. It converts the endpoints in path file records from 3 character location codes to 4 character switch codes. Where more than one switch resides at a location, all possible combinations are produced. This step is necessary in order to correlate physical paths to trunk groups (which have switch code endpoints) in the `mat_trk` module.

Call Syntax `clli3_4 <filename>`

where `<filename>` specifies the name of the input file containing a list of all other input and output files

example `clli3_4 MCIfiles.fy94`

Input Files

list file — This file simply contains the names of the two input files and one output file. File names are limited by `clli3_4` to a length of 50 characters.

format

line1:	<input path file name>
line2:	<switch location file>
line3:	<output path file name>

input path file — Each record in this file, specifies the physical transmission path between a pair of switches. A physical transmission path is defined by the series of spans (from none for collocated switches to a limit of 662) that connect a switch pair. Spans are identified by indexes that point to the appropriate record number in the span file.

format <switch CLLI A>, <switch CLLI Z>, <span1>, <span2>, ... , <span n>
(c11, 1x, c11, 1x, i6, i6, ... , i6)

example
```
AST         NYC          3045   3042   7579    237
```

switch location file — This file contains each MCI 4 character switch code, followed by its 3 character location code. The location code identifies the building in which the switch is housed.

format <switch code>, <location code>
(c4, 1x, c3)

example
```
AST1 AST
AST2 AST
NYC1 NYC
```

Output Files

output path file — This file is in the same format as the input path file, except that each path endpoint has been mapped from a location code to all possible combinations of switch codes.

format same as input path file

example	AST1	NYC1	3045	3042	7579	237
	AST2	NYC1	3045	3042	7579	237

Includes

`<stdio.h>`
`<string.h>`
`"fileio.c"` See Appendix B

Constants

`MAXLINE 4000` maximum number of characters in a path file record

Global Variables

file	`*pathfile, *locfile, *outfile, *filelist`		pointers to the input and output files
integer	`num_nodes`		counts the number of switches read from the switch location file
character	`line[MAXLINE]`		holds a line read from the input path file
structure	`mapp_struct`	`mapp[]`	used to hold each record in the switch location file
	with field character	`mapp[].three`	holds MCI three character location code read in from the switch location file
		`mapp[].four`	holds MCI four character switch code read in from the switch location file

Component Functions

`openfiles()`	opens input and output files
`readmappings()`	reads in the list of switch and location codes from the switch location file
`createpathfile()`	maps the path endpoints from 3 character location codes to 4 character switch codes
`closefiles()`	closes input and output files

Function Tree

```
          ┌─openfiles()
          ├─readmappings()
main()────┤
          ├─createpathfile()
          └─closefiles()
```

Algorithmic Description

This module addresses a shortfall in MCI data only. It converts the endpoints in path file records from 3 character location codes to 4 character switch codes. Where more than one switch resides at a location, all possible combinations are produced. For example, a path between two locations which house 2 and 3 switches respectively would be mapped to the 6 possible switch-to-switch paths. This module is required for the subsequent running of the `mat_trk` module, which correlates the logical trunk group file with the physical path file based on the switch endpoints of each record.

The `main()` routine passes the `<filename>` argument (the name of the file list file) to `openfiles()`. `Openfiles()` opens all input and output files whose names are contained in the file list. Then `readmappings()` is called to read the contents of the switch location file into the `mapp[]` structure. `Main()` calls `createpathfile()` to apply the location-to-switch endpoint mapping to each record of the input path file, thereby creating the output path file. `Closefiles()` is called to close all files before the module terminates.

Inputs			
character	`files`	string containing the name of the file that lists the three input/output files	
Outputs			
global	file	`*filelist`	points to the file whose name is contained in the `files` string
		`*pathfile`	points to the input physical transmission path file, using location endpoints
		`*locfile`	points to a file that lists the MCI 4 character switch codes and corresponding3 character location codes
		`*outfile`	filtered version of path file using switch endpoints instead of location endpoints.

returns no formal values are returned

Purpose To open path and location input files and the output path file.

Called By `main()`

Calls To none

Local Variables

character `tempfile[50]` this variable is used temporarily to hold the name of the next file to be opened and read in from the list of files in `filelist`

Global Variables none

Algorithmic Description

This function opens the file whose name is stored in the string `files`, setting a file pointer, `filelist`. `Filelist` contains a list of all the input/output files to be opened in the following order: `pathfile`, `locfile` and `outfile`. This function then proceeds to open each of these files, in the order they are read in from `filelist`, assigning them to the matching file pointers.

Errors encountered during any file opening operation result in an error message being printed to the screen, and termination of the module.

Inputs	none; operates on global variables		
Outputs			
global	mapp_struct structure	`mapp[]`	holds the records from the switch location file
	with fields character	`mapp[].three` `mapp[].four`	holds MCI three-character location code holds MCI four-character switch code
returns	no formal values are returned		

Purpose To read the list of switch codes and corresponding location codes from the switch location file, into the `mapp[]` structure.

Called By `main()`

Calls To none

Local Variables

integer	`i`	general loop count variable

Global Variables

integer	`num_nodes`	the total number of switch records read
file	`*locfile`	points to the switch location file

Algorithmic Description This function reads the switch location file line by line, loading each location code into the `mapp[].three` field and each switch code into the `mapp[].four` field, and setting `num_nodes` to the total number of records in the switch location file.

Inputs none; operates on global variables

Outputs

returns no formal values are returned; outputs are written directly to the output file

Purpose This function maps the path endpoints from three character location codes to all possible combinations of four character switch codes.

Called By `main()`

Calls To none

Local Variables

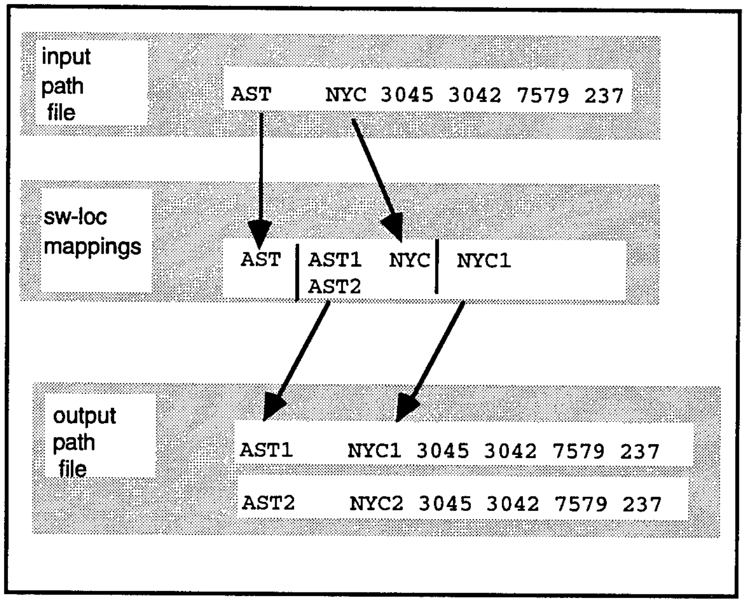
integer	`i`	general loop count variable
	`len`	used to hold the string length of a `pathfile` line
	`found1=0`	a toggle used to indicate whether `sw1` was found in the `mapp_struct` structure
	`found2=0`	a toggle used to indicate whether `sw2` was found in the `mapp_struct` structure
	`pos1`	the index of `sw1` within the `mapp_struct` structure
	`pos2`	the index of `sw2` within the `mapp_struct` structure
	`first2`	temporarily holds the value of `pos2`
character	`sw1`	the originating endpoint of the current path
	`sw2`	the terminating endpoint of the current path

Global Variables

integer	`num_nodes`	the number of records in the `pathfile`	
character	`line[]`	a line of input from the `pathfile`	
structure	`mapp_struct`	`mapp[]`	used to hold each record from the switch location file
	with fields character	`mapp[].three`	three character location code
		`mapp[].four`	four character switch code

Algorithmic Description This function processes the path file, record by record, loading the three character location for the originating and terminating endpoints into the string variables `sw1` and `sw2`. Next this function searches the `mapp[]` structure for each of the four character switch CLLI codes, at locations `sw1` and `sw2`. A path record is printed to the output file for every combination of switch CLLI code endpoints at locations `sw1` and `sw2`

The following example illustrates the dataflow performed by createpathfile():



Inputs none; operates on global variables

Outputs

global	file	`*pathfile`	points to a file that contains information on the physical transmission paths between pairs of switches
		`*locfile`	points to a file that contains the list of MCI 4 character switch codes and 3 character location codes
		`*outfile`	filtered version of path file

returns no formal values are returned

Purpose To close path and location input files and the output file.

Called By `main()`

Calls To none

Local Variables none

Global Variables none

Algorithmic Description This function closes the files whose names are pointed to by the following pointers: `pathfile`, `locfile` and `outfile`, in the order given.

3.10 mkpath: Make Path module

Purpose This module consolidates the trunk and path files output from `mat_trk`. Switch CLLI codes are replaced with index numbers that reference the switch list.

Call Syntax `mkpath <filename>`

where `<filename>` specifies the name of the input file containing a list of all other input and output files

example `mkpath MCIfiles.fy94`

Input Files

<u>list file</u> This file simply contains the names of the three input files and one output file to be used by `mkpath`. File names are limited by `mkpath` to a length of 50 characters.

format
line1: <path file name>
line2: <switch file name>
line3: <trunk file name>
line4: <output file name>

<u>switch file</u> This file contains the list of codes for IEC backbone switches.

format <IEC switch CLLI code>
(c11)

<u>trunk file</u> The trunk file specifies the IEC switch CLLI codes of the trunk endpoints, the number of trunks in the A to Z direction (or all bi-directional trunks), and the number of trunks in the Z to A direction (only used for one-way trunk groups). Each record is in 1-to-1 correspondence with the path file. That is, the number of trunks in the n^{th} trunk file record traverse the transmission path specified by the n^{th} path file record.

format <switch CLLI A>, <switch CLLI Z>, <A->Z trunk quantity>, <Z->A trunk quantity>
(c11, 1x, c11, 1x, i4, 1x, i4)

<u>path file</u> Each record in this file, created by the `mat_trk` module, specifies the physical transmission path between a pair of switches. A physical transmission path is defined by the series of spans (from none for collocated switches to a limit of 662) that connect a switch pair. Spans are identified by indexes that point to the appropriate record number in the span file.

format <switch CLLI A>, <switch CLLI Z>, <span1>, <span2>, ... , <span n>
(c11, 1x, c11, 1x, i6, i6, ... , i6)

Output Files

<u>output trunk/ path file</u> The output trunk/path file combines the logical trunk and physical path records into a single record format that specifies trunk quantities

per physical path. The format specifies the index numbers of the IEC switch endpoints, the number of trunks in the A to Z direction (or all bi-directional trunks), the number of trunks in the Z to A direction (only used for one-way trunk groups), and a series of span index numbers that define the physical transmission path.

format <switch index A>, <switch index Z>, <A->Z trunk quantity>, <Z->A trunk quantity>, <span1>, <span2>, ... , <span n>
(i6, i6, i6, i6, [spans:] i6, i6, ... , i6)

Includes

`<stdio.h>`
`<string.h>`
`"fileio.c"` See Appendix B

Constants

`MAXPATH`	`4000`	maximum number of characters in a path file record
`CLLI_LNG`	`12`	length of a switch CLLI code, including terminating null character
`MAXSW`	`200`	maximum number of records in the switch file

Global Variables

file `*pathfile, *switchfile, *trunkfile, *filelist, *outfile`
pointers to the input and output files

integer `num_nodes` the number of records read in from the switch file

character `line[MAXPATH]` used to hold a line of input from the path file
`swline[MAXSW]` used to hold a line of input from the switch file

structure `sw[MAXSW]` of type `switch_struct`
with fields:
character `sw[].clli` used to hold the list of switch CLLI codes

Component Functions

`openfiles()`	opens input and output files
`readswitches()`	reads in the list of switches from the switch file
`createpathfile()`	combines path and trunk records into a single output record
`closefiles()`	closes input and output files

Function Tree

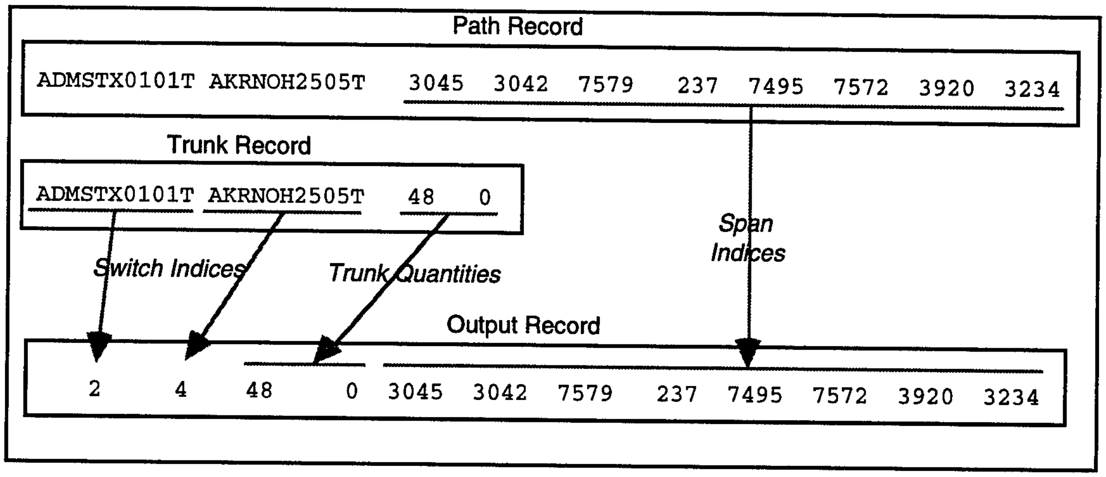
```
          ┌─openfiles()
          │
          ├─readswitches()
main() ───┤
          ├─createpathfile()
          │
          └─closefiles()
```

Algorithmic Description

This module performs the simple task of consolidating the trunk and path files output from `mat_trk` and replacing switch CLLI codes with index numbers that reference the switch list. Because the trunk and path file records are already in one-to-one correspondence, `mkpath` essentially concatenates the two records into a single output record format.

The program begins by calling `openfiles()` to open the input and output files. Next, `readswitches()` is called to load the switch list into the `sw[]` structure. Then, `createpathfile()` is called to perform the main algorithm. This routine reads in a path record and a trunk record. It parses the switch endpoint CLLI codes from the path record, and searches through the switch structure to determine the corresponding switch index numbers. The output record is written using these index numbers, along with the trunk quantity and transmission path information. Note that no processing or filtering is performed on either the trunk or path records.

The following example shows the data flow through mkpath that maps corresponding path and trunk records into a single output record:



Inputs			
character	`files`	string containing the name of the file that lists the five input/output files	
Outputs			
global	file	`*filelist`	points to the file whose name is contained in the `files` string
		`*pathfile`	points to a file that contains information on the physical paths and switches
		`*switchfile`	points to a file that contains information on the backbone network switches
		`*trunkfile`	points to a file that contains information on the logical trunks
		`*outfile`	filtered version of path file
returns	no formal values are returned		

Purpose To open path, trunk and switch input files and the path output file.

Called By `main()`

Calls To none

Local Variables

character	`tempfile[50]`	this variable is used temporarily to hold the name of the next file to be opened and read in from the list of files in `filelist`

Global Variables none

Algorithmic Description

This function opens the file whose name is stored in the string `files`, setting a filepointer to `filelist`. `Filelist` contains a list of all the input files to be opened in the following order: `pathfile`, `switchfile`, `trunkfile`, and `outfile`. This function then proceeds to open each of these files, in the order they are read in from `filelist`, assigning them to the matching filepointers.

Errors encountered during any file opening operation result in an error message being printed to the screen, and termination of the module.

Inputs none; operates on global variables

Outputs

global

switch_struct structure		sw[]	used to hold each record in the switch file
with field:s			
	character	sw[].clli	holds switch endpoints read in from the switchfile
	integer	num_nodes	used to count the number of records in the switchfile
	file	*switchfile	points to the switch file

returns no formal values are returned

Purpose To read the list of switch CLLI codes from the switch file into the `sw[]` structure.

Called By `main()`

Calls To none

Local Variables

integer	i	general loop count variable

Global Variables

character	swline[]	used to hold a line of input from the switch file

Algorithmic Description

This function reads the switch file line by line, loading each CLLI code into the structure `sw[]` and setting `num_nodes` to the total number of switches in the switch file.

Inputs none; operates on global variables

Outputs

returns no formal values are returned; outputs are written directly to the output file

Purpose This function consolidates the trunk and path files output from `mat_trk`. Switch CLLI codes are replaced with index numbers that reference the switch list.

Called By `main()`

Calls To none

Local Variables

integer	`i, j`	general loop count variables
	`len`	used to hold the string length of a `pathfile` line
	`trunksizeA`	the number of trunks in the A ->Z direction
	`trunksizeZ`	the number of trunks in the Z ->A direction
character	`temp1[]`	used to hold the originating endpoint of a path
	`temp2[]`	used to hold the terminating endpoint of a path
	`trkline[]`	temporarily holds a record from the `trunkfile`

Global Variables

integer	`num_nodes`	used to hold the number of records in the `switchfile`
character	`line[]`	used to hold a line of input from the `pathfile`

Algorithmic Description

This function processes the trunk and path files, record-by-record, combining the information into a single output file record. For each line (record) in the trunk file, this function parses the number of trunks in the A->Z and the Z->A directions, and stores these fields in `trunksizeA` and `trunksizeZ`, respectively. For each line (record) in the path file, the function parses the originating and terminating path endpoints, and stores these fields in `temp1` and `temp2`, respectively.

This function then derives indices to replace switch CLLI codes by locating the position of `temp1/temp2` within `sw[]`. A combined trunk/path record is written to the output file as: *switch 1 index, switch 2 index, trunksize of A, trunksize of Z, spans* (repeated from input path record).

Inputs	none; operates on global variables		
Outputs global	file	`*trunkfile`	points to a file that contains information on the logical trunks
		`*switchfile`	points to a file that contains information on the backbone network switches
		`*pathfile`	points to a file that contains information on the physical paths and switches
		`*outfile`	combined and indexed trunk and path file
returns	no formal values are returned		
Purpose	To close trunk, switch and path input files and the output file.		
Called By	`main()`		
Calls To	none		
Local Variables	none		
Global Variables	none		

Algorithmic Description

This function closes the files whose names are pointed to by the following pointers: `trunkfile`, `switchfile`, `pathfile` and `outfile`, in the order given.

3.11 damage: Monte Carlo Damage module

Purpose For every network asset, this program generates a number of Monte Carlo damage values. A damage value is either a 0 to indicate equipment failure, or a 1 to indicate survival. Two general categories of assets are damaged: nodes and spans. Damage is based on equipment type. Each type has a cumulative distribution function (CDF) which defines the equipment's probability of failure. Specifically, this module is designed to apply electromagnetic pulse (EMP) damage to network facilities.

Call Syntax `damage -i <filename> [options]`

mandatory:

`-i <filename>`	specifies the name (`<filename>`) of the input keyfile which contains a list of the run parameters and input/output asset filenames

options:

`-a`	adds a live damage vector to the output file
`-h`	uses 10dB shielding CDF's for AT&T Series G fiber damage rather than the default 6dB shielding (EMP specific)
`-p`	uses AT&T Series G fiber CDF's without power supply failure for damage rather than the default 6dB shielding (EMP specific)
`-s <integer>`	sets the random number stream (1-15)
`-?`	user help--prints call syntax and exits without running

example `damage -i ATTassetts.key -s 6 -a -h` (spaces optional)

Input Files

<u>keyfile</u> This file contains the input parameters and asset filenames to be used by `damage`. The parameters specify the type of damage to perform (EMP vs. fallout radiation), the number of damage vectors to generate in each of three intensity ranges (low, medium, and high), whether to include the effects of switch upset, and the name of the CDF file. CDF and asset file names are limited by `damage` to a length of 80 characters.

format

line1:	<damage type> either `EMP` or `FR` (fallout radiation)
line2:	<number of low damage vectors>,<medium>,<high> (i, 1x, i, 1x, i)
line3:	<switch upset toggle> either `UPSET_ON` or `UPSET_OFF`
line4:	<CDF file name>
line5+:	<node/span indicator (`N/S`)>,<asset file>,<damage file> (c1,1x,c,1x,c) *example node file:* `N switchlist switchlist.dmg` *example span file:* `S spanlist spanlist.dmg`

<u>node file</u> This file contains a list of switches. Each line contains an 11-character CLLI code followed by a 3-character equipment code (see algorithm description for details). Any information following the equipment type is ignored.

format: <switch CLLI code>,<equipment code>
(c11, 1x, c3)

__span file__ This file contains the list of spans. Each line contains 2 11-character CLLI codes (the two span endpoints), a 2-character equipment code (see algorithm description for details), and V-H coordinates for each endpoint. Any information following the coordinates is ignored.

format: <CLLI A>,<CLLI Z>,<equipment code>,<V-coord A>,<H-coord A>,
<V-coord B>,<H-coord B>
(c11, 1x, c11, 1x, c2, 1x, i5, 1x, i5, 1x, i5, 1x, i5)

__CDF file__ This file contains the data points for the CDF curves for all of the node and span equipment to be damaged. Each CDF consists of 100 data points listed 5 per line (i.e., 20 lines per curve). The file contains the y-values of the curve for $0<x\leq 1$ in 0.1 increments. There are no divisions or indicators between CDF's. The identity of each CDF is given by a key in the damage code (described in the Constants section below).

format <data point>,<data point>,<data point>,<data point>,<data point>
(e13.7, 1x, e13.7, 1x, e13.7, 1x, e13.7, 1x, e13.7)

Output Files

__output node file__ Each line of this file contains a CLLI code followed by a number of damage values (0 indicates equipment, 1 indicates survival). Each line contains the same number of damage values. This number of specified in the keyfile.

format <switch CLLI A>, <damage values 1,2,3,...,n>
(c11, 1x, i1, i1, i1,...,i1)

__output span file__ Each line of this file contains the two endpoint CLLI codes followed by a number of damage values (0 indicates equipment, 1 indicates survival). Each line contains the same number of damage values. This number of specified in the keyfile.

format <CLLI A>, <CLLI B>, <damage values 1,2,3,...,n>
(c11, 1x, c11, 1x, i1, i1, i1,...,i1)

Includes

`<stdio.h>`	
`<math.h>`	
`"fileio.c"`	See Appendix B
`"/user/gretchen/waglib/waglib.h"`	Random number generator

Constants

MAX_CDF	75	maximum number of CDF curves in a CDF file
MAX_SUP	25	maximum number of supplemental CDF curves in a CDF file
FALSE	0	logical false
NONE	(-1)	no curve selected
ANASWT	0	CDF index: generic analog switch
DIGSWT	3	CDF index: generic digital switch
L4COAX	6	CDF index: L4 coaxial transmission system
T1OFFC	9	CDF index: T1 carrier with office repeater
T1LINE	12	CDF index: T1 carrier with line repeater
FT3SWT	15	CDF index: AT&T fiber carrier with FT3C terminal
FT3RPT	18	CDF index: AT&T fiber carrier with FT3C repeater
MWTD2	21	CDF index: TD-2 analog microwave
ATT4ES	24	CDF index: AT&T 4ESS switch (damage curve)
FOR140	27	CDF index: Alcatel fiber carrier with R-R140 repeater
DMS100	30	CDF index: NT DMS-100 switch (damage curve)

DMSUPS	33	CDF index: NT DMS-100 switch (upset curve)
ATT4EU	36	CDF index: AT&T 4ESS switch (upset curve)
ATT5ES	39	CDF index: AT&T 5ESS switch (damage curve)
ATT5EU	42	CDF index: AT&T 5ESS switch (upset curve)
FTGDMG	45	CDF index: AT&T Series G fiber (damage curve)
FTGUPS	48	CDF index: AT&T Series G fiber (upset curve)
FTG6D	45	CDF index: Series G with 6dB shielding (damage)
FTG6U	54	CDF index: Series G with 6dB shielding (upset)
FD565	57	CDF index: NTI FD-565 fiber terminal
FTG10D	60	CDF index: Series G with 10dB shielding (damage)
FTG10U	63	CDF index: Series G with 10dB shielding (upset)
FTGPSD	45	CDF index: Series G ignoring damage to power supply (damage)
FTGPSU	45	CDF index: Series G ignoring damage to power supply (upset)

SUPFTG6D	0	CDF index: Series G (6dB) supplemental data points
SUPFTG6U	3	CDF index: Series G (6dB) supplemental data points
SUPFTG10D	6	CDF index: Series G (10dB) supplemental data points
SUPFTG10U	9	CDF index: Series G (10dB) supplemental data points
SUPFD565	12	CDF index: FD-565 supplemental data points
SUPFTGPSD	15	CDF index: Series G (power supply) supplemental data points
SUPFTGPSD	15	CDF index: Series G (power supply) supplemental data points

RAD4ES	0	CDF index: AT&T 4ESS fallout radiation curves
RAD5ES	0	CDF index: AT&T 5ESS fallout radiation curves
RADFOR	3	CDF index: fiber optic fallout radiation curves

Local Variables

Variables local to `main()`:

integer	err	an error flag indicating a problem with the command line arguments
	itog	a flag indicating that the -i command line option is set
	stog	a flag indicating that the -s command line option is set
	htog	a flag indicating that the -h command line option is set
	ptog	a flag indicating that the -p command line option is set
	VA	vertical coordinate of node A
	HA	horizontal coordinate of node A
	VZ	vertical coordinate of node Z
	HZ	horizontal coordinate of node Z
	i	an index variable
	j	an index variable
	count	an output counter
	optind	the number of a single command line arguments
	argc	the number of command line arguments
character	input_file[50]	used to read a file name from a prompt
	ch	a single character
	line[100]	holds a single input line from a file
	temp[6]	used to parse a line
	clliA[12]	the CLLI code for node A
	clliZ[12]	the CLLI code for node Z
	equip[4]	an equipment code
	*optarg	a string containing a single command line argument
	**argv[]	an array containing all of the command line arguments
file pointer	fptr	file pointer
	inptr	input file pointer
	outptr	output file pointer

structure	`*ptrlist` of type `flist`		
	with fields:		
	character	`ptrlist.in`	input file
		`ptrlist.out`	output damage file
	pointer	`ptrlist.ptrnext`	points to next item in `ptrlist` list

Global Variables

integer	`MODE`	damage mode (0=EMP, 1=fallout radiation)
	`UPSET`	switch upset toggle (0=off, 1=use upset curves)
	`LIVE_VEC`	toggle to add live vector to output (0=off, 1=on)
	`NODE_DMG`	toggle to indicate if node file is present (1) or absent (0)
	`SPAN_DMG`	toggle to indicate if span file is present (1) or absent (0)
	`FTG_DMG_PTR`	pointer to the Series G damage CDF curves in use (e.g., 6dB)
	`FTG_UPS_PTR`	pointer to the Series G upset CDF curves in use (e.g., 6dB)
	`tot_iter[3]`	number of Monte Carlo iterations (low, medium, and high)
	`tot_cdf`	number of CDF curves loaded
	`PSTOG`	toggle to use Series G curves without power supply damage
	`num_nodes`	number of nodes read for damage
	`num_spans`	number of spans read for damage
	`node_stats[6]`	counts the number of each node category
	`span_stats[11]`	counts the number of each span category
	`length_count`	total number of spans included for average length
	`span_dmg_stats[11][2][3]`	span damage stats by category (out of 11) and damage level (of 3)
	`node_dmg_stats[6][2][3]`	node damage stats by category (out of 6) and damage level (of 3)
	`stream_num1`	random number stream #1
	`stream_num2`	random number stream #2
	`Fcount`	fiber diagnostic variable
double	`Fsum`	fiber diagnostic variable
	`cdf_table[MAX_CDF][100]`	100-point CDF curves
	`supp_cdf_table[MAX_SUP][100]`	100-point supplemental CDF curves
	`length_sum[11]`	sum of span lengths (by category)
	`length_sumsqr[11]`	sum of span lengths squared (by category)
character	`cdf_file[81]`	name of the main CDF file
	`analog_switch_types[]`	contains the 3-character switch equipment codes which are assigned to the generic analog switch CDF curve for damage
	`digital_switch_types1[]`	contains half of the 3-character switch equipment codes which are assigned to the generic digital switch CDF curve for damage
	`analog_switch_types2[]`	contains half of the 3-character switch equipment codes which are assigned to the generic digital switch CDF curve for damage
	`nt_digital_types[]`	

contains the 3-character switch equipment codes which are assigned to the DMS-100 CDF curve for damage

structure

`nodefiles` of type `flist`
with fields:

character	`nodefiles.in`	input node file
	`nodefiles.out`	output damage file
pointer	`nodefiles.ptrnext`	points to next item in `nodefiles` list

`spanfiles` of type `flist`
with fields:

character	`spanfiles.in`	input span file
	`spanfiles.out`	output damage file
pointer	`spanfiles.ptrnext`	points to next item in `spanfiles` list

`EQ_list,EQ_ptr` of type `EQ`
with fields:

character	`EQ_list.eq_type[4]`	equipment code
integer	`EQ_list.freq`	number of occurrences
pointer	`EQ_list.ptrnext`	points to next item in `EQ_list` list

Component Functions

`LoadKey()`	opens and reads the keyfile
`TallyUnknown()`	counts the number of times an unknown equipment type is found in an asset file
`LoadCDF()`	loads a CDF file into memory
`LoadSuppCDF()`	loads supplemental CDF data into memory
`detprb()`	picks a random point from a CDF curve and returns a probability associated with that point
`Survive()`	returns a random survival value based on a CDF curve
`DmgNode()`	generates a series of damage vectors for one node
`DmgSpan()`	generates a series of damage vectors for one span
`PrintNodeStats()`	prints summary node statistics
`PrintSpanStats()`	prints summary span statistics

Function Tree

```
                ┌─LoadKey()
                ├─LoadCDF()
                ├─LoadSuppCDF()
                │                      ┌─TallyUnknown()
       main()───┼─DmgNode() ───────────┤
                │                      └─Survive() ─ detprb()
                ├─PrintNodeStats()
                │                        ┌─TallyUnknown()
                ├─DmgSpan() ─────────────┼─Survive() ─ detprb()
                └─PrintSpanStats()       └─detprb()
```

Algorithmic Description

This module uses equipment survivability data to determine probabilistically the survival or failure of network equipment. This module was designed to interpret CDF curves representing the probability of damage due to EMP effects. Tests have been performed on a number of switching and transmission systems. For each equipment type, physical survivability CDF curves were calculated for each of the three EMP stress levels (low, medium, and high). In addition, switch upset CDF curves were calculated for four equipment types. The constants section above details the available EMP CDF curves.

This set of equipment represents a major portion, although not a comprehensive set, of the equipment types employed in the PSN. There are no EMP test data for some network equipment types. Rather than assume they survive EMP damage, equipment types that have not been tested are assigned the survivability of the tested equipment type that they most closely resemble.

The general procedure for testing equipment failure (referred to as the "CDF Test") is the following:

1) Pick a random number, Y (uniform distribution, 0-1).
2) Find the x-value, X, on the CDF curve corresponding to the y-value, Y.
3) Pick a second random number, A (uniform distribution, 0-1).
4) If A≤X then the equipment survives the CDF Test, otherwise it fails.

EMP node damage is the simplest to assess. To survive EMP damage, a node need only pass a single CDF Test with the CDF curve to which the equipment has been assigned. To survive switch upset, a node must pass the damage CDF Test plus an additional test with the assigned EMP upset curve. Switch upset does not apply to equipment assigned to the Generic Analog CDF curve.

EMP span damage is more complex. In general, a minimum of two CDF Tests are required to determine survival/failure—one CDF Test for each endpoint. However, long spans which require intermediate repeater equipment will require additional CDF Tests for the additional equipment. The assumed spacing between repeaters varies by equipment type: 23 miles for optical fiber, 26 miles for microwave, and 1 mile for T1.

Two additional exceptions apply to span damage. First, T1 links longer than 50 miles are assumed to be data errors, and are assigned to the Series G optical fiber CDF curve. Second, LEC spans are assumed to be 80% fiber and 20% T1.

A final consideration for Series G fiber damage. Several CDF curves are available for this span type based on a number of assumptions. The default Series G curves assume 6dB shielding around the repeaters. Using the `-h` option, the user may assess Series G damage using the 10dB shielding curves. Additionally, the `-p` option assess Series G damage using curves which do not consider damage to the equipment power supplies. Options `-h` and `-p` may not be used together.

After the command line arguments have been parsed and interpreted, `main()` executes in the following order:

1) Calls `LoadKey()` to open and interpret the keyfile.

2) Calls `LoadCDF()` to load the main CDF file.

3) Calls `LoadSuppCDF()` to load the supplemental CDF file. this file contains 100 additional data points between 0.9 and 1.0 for certain equipment.

4) For each node record in the first node file, calls `DmgNode()` to produce damage vectors for that node. `DmgNode()` writes the output data to an output damage file.

5) Calls `PrintNodeStats()` to get summary damage statistics for the node file.

6) Prints a summary of all unknown switch equipment types found in the file.

7) Repeats Steps 4 through 6 for each node file specified by the keyfile.

8) For each span record in the first span file, calls `DmgSpan()` to produce damage vectors for that span. `DmgSpan()` writes the output data to an output damage file.

9) Calls `PrintSpanStats()` to get summary damage statistics for the span file.

10) Repeats Steps 8 and 9 for each span file specified by the keyfile.

Inputs			
file	`*fptr`		a file pointer to the keyfile (already opened)
Outputs			
globals	integer	`MODE`	the damage mode (0=EMP, 1=fallout radiation)
		`tot_iter[3]`	the number of low, medium, and high iterations
		`UPSET`	toggle for switch upset (0=off, 1=on)
	character	`cdf_file[]`	the name of the main CDF file

structures `nodefiles` and `spanfiles` of type `flist`
with fields:

character	`in[]`	input asset file
	`out[]`	output damage file
pointer	`*ptrnext`	points to next item in list

Purpose To load run parameters from the keyfile.

Called By `main()`

Calls To none

Local Variables

integer	`count`	holds the current line number in the keyfile
	`flag`	error flag
character	`line[80]`	holds an entire line from the keyfile
	`infile[81]`	temporarily holds the name of an input asset file
	`outfile[81]`	temporarily holds the name of an output damage file
	`type`	a single character holding the type of asset file (N=node, S=span)
pointer to structures	`*ptrnew`	points to a new `flist` entry to be inserted into a list
	`*ptrlastspan`	points to the end of the `spanfiles` list
	`*ptrlastnode`	points to the end of the `nodefiles` list

Global Variables none

Algorithmic Description

This function reads each line of the keyfile and sets run parameters according to entries in the keyfile. All of the run parameters are held in global variables. The line-by-line structure of the keyfile is shown in the `damage` module description.

Inputs

file	`*fptr`		a pointer to the main CDF file (already opened)

Outputs

globals	double	`cdf_table[MAX_CDF][100]`	100-point CDF curves
		`tot_cdf`	the number CDF curves loaded

Purpose To load CDF curves from the main CDF file.

Called By `main()`

Calls To none

Local Variables

integer	`i`	an index variable
	`count`	the line number of the current CDF curve
	`tog`	toggle to indicate that the 50% point has been passed on the curve
	`pcount`	cycles from 0 to 2 to indicate low, medium, or high stress levels
character	`line[100]`	holds an entire line from the CDF file
	`temp[15]`	used to parse a CDF data point from `line`

Global Variables none

Algorithmic Description

This function parses each line of a CDF file to extract 5 y-values from a 100-point CDF curve. The curves are assumed to be grouped by the three EMP stress levels (low, medium, and high). The structure of the CDF file is described in the `damage` module.

3.11.3 LoadSuppCDF function

Inputs

file `*ftg_fd565.cdf_supp`
the supplemental CDF file for Series G and FD-565

Outputs

globals double `supp_cdf_table[MAX_SUP][100]`
100-point supplemental CDF curves

Purpose To load supplemental CDF curves from the supplemental CDF file.

Called By `main()`

Calls To none

Local Variables

file	`*fptr`	pointer to the supplemental CDF file
integer	`i`	an index variable
	`count`	the line number of the current CDF curve
	`tog`	toggle to indicate that the 50% point has been passed on the curve
	`pcount`	cycles from 0 to 2 to indicate low, medium, or high stress levels
character	`line[100]`	holds an entire line from the CDF file
	`temp[15]`	used to parse a CDF data point from `line`

Global Variables none

Algorithmic Description

This function parses each line of the supplemental CDF file to extract 5 y-values from a 100-point supplemental CDF curve. These points correspond to CDF x-values between 0.9 and 1.0. The curves are assumed to be grouped by the three EMP stress levels (low, medium, and high). The structure of the supplemental CDF file is identical to the main CDF file (described in the `damage` module).

Inputs

character	`clliA[]`	the 11-character switch CLLI code
	`equip[]`	the 3-character switch equipment code
integer	`MODE`	the damage mode (0=EMP, 1=fallout radiation)
	`UPSET`	the switch upset mode (0=off, 1=on)
	`LIVE_VEC`	the toggle to add an output live vector (0=off, 1=on)
	`tot_iter[3]`	the number of vectors to generate (low, medium, high)
file	`*outptr`	pointer to the output damage file

Outputs

global	integer	`node_stats[6]`	counts the number of nodes in each node category
		`node_dmg_stats[6][2][3]`	node damage stats by category (out of 6) and damage level (out of 3) where the middle subscript allows for holding live and damage totals

Purpose Generates damage vectors for a single node.

Called By `main()`

Calls To
`TallyUnknown()`
`Survive()`

Local Variables

integer	`column`	an index into the CDF table pointing to the damage curve
	`upset_col`	an index into the CDF table pointing to the upset curve
	`bin`	the EMP stress level (0=low, 1=medium, 2=high)
	`it`	iteration loop variable
	`surviv`	node equipment survival (1) or failure (0)

Global Variables none

Algorithmic Description

This function generates the number of damage vectors specified by `tot_iter[]`. Damage is based on the type of node equipment and the level of EMP damage being assessed. Damage may be based on EMP or fallout radiation curves (based on `MODE`), and may include the effects of switch upset (based on `UPSET`). Finally, a live damage vector may be added to the beginning of the output damage stream (based on `LIVE_VEC`).

For a given node, the following procedure is followed:

1) Assign damage and upset CDF curves to the node based on the input equipment type. If the equipment code is "unknown," then call `TallyUnknown()` with the code. Assume equipment is of type 5ESS.

2) Call `Survive()` with the base CDF curve (`column`) plus the stress level (`bin`). `Survive()` returns 1 (survive) or 0 (fail).

3) If the node survives Step 2 and `UPSET` is on, then call `Survive()` with the upset CDF curve (`upset_col`) plus the stress level (`bin`). `Survive()` returns 1 (survive) or 0 (fail).

4) If the node survives both Steps 2 and 3, then print a '`1`' to the output damage file. Otherwise, print a '`0`'.

5) Tally up the damage stats in `node_dmg_stats[]`.

6) Repeat Steps 2 through 5 for the number of iterations specified by `tot_iter[]` for the current EMP stress level.

7) Repeat Step 6 for each EMP stress level.

Inputs none

Outputs none

Purpose To print summary statistics for each node type.

Called By `main()`

Calls To none

Local Variables none

Global Variables

integer	`node_stats[6]`	counts the number of nodes in each node category
	`node_dmg_stats[6][2][3]`	node damage stats by category (out of 6) and damage level (out of 3) where the middle subscript allows for holding live and damage totals

Algorithmic Description

This function prints out node statistics. For each category equipment (e.g., 4ESS), the total number of nodes in the category are printed, as well as the percentage of all nodes that that category accounts for. Finally, the percentage of node damaged at each EMP stress level (low, medium, high) is shown.

Inputs

character	cllia[]	the 11-character originating CLLI code
	clliZ[]	the 11-character terminating CLLI code
	equip[]	the 2-character switch equipment code
integer	MODE	the damage mode (0=EMP, 1=fallout radiation)
	UPSET	the switch upset mode (0=off, 1=on)
	LIVE_VEC	the toggle to add an output live vector (0=off, 1=on)
	tot_iter[3]	the number of vectors to generate (low, medium, high)
	VA	the V-coordinate of clliA
	HA	the H-coordinate of clliA
	VZ	the V-coordinate of clliZ
	HZ	the H-coordinate of clliZ
file	*outptr	pointer to the output damage file

Outputs

global	integer	span_stats[11]	counts the number of spans in each span category
		span_dmg_stats[11][2][3]	span damage stats by category (out of 11) and damage level (out of 3) where the middle subscript allows for holding live and damage totals

Purpose Generates damage vectors for a single span.

Called By `main()`

Calls To

```
TallyUnknown()
Survive()
detprb()
```

Local Variables

integer	column	an index into the CDF table pointing to the damage curve
	upset_col	an index into the CDF table pointing to the upset curve
	bin	the EMP stress level (0=low, 1=medium, 2=high)
	it	iteration loop variable
	surviv	node equipment survival (1) or failure (0)
	i	in index variable
	n	the number of times a CDF Test must be repeated
	equip_type	a code number indicating the equipment category
double	D	the length of a span
	fo_prob	the survival probability of a LEC fiber span
	t1_prob	the survival probability of a LEC T1 span
	prob1	an aggregate survival probability of a LEC span
	prob2	a random number

Global Variables none

Algorithmic Description

This function generates the number of damage vectors specified by `tot_iter[]`. Damage is based on the type of span equipment, the length of the span, and the level of EMP damage being assessed. Damage may be based on EMP or fallout radiation curves (based on `MODE`), and may include the effects of switch upset (based on `UPSET`). Finally, a live damage vector may be added to the beginning of the output damage stream (based on `LIVE_VEC`).

For a given span, the following procedure is followed: (NOTE: Pages 9-14 of Reference 5 contains a complete description of the damage procedure.)

1) Calculate the length of the span.

2) Decode the equipment type and assign a code to variable `equip_type`. This code may reflect a change in equipment type (e.g., long T1s are assumed to be Series G optical fiber). If the equipment code is "unknown," then call `TallyUnknown()` with the code. Assume the equipment is of Series G optical fiber.

3) Accumulate average length statistics for the equipment type.

4) Assign damage and upset CDF curves to the span based on `equip_type`.

5) For most equipment types, determine the number of repeaters based on the span length.

6) Call `Survive()` for each endpoint and for each repeater.

7) If the span survives all of the CDF Tests in Step 6 and `UPSET` is on, then call `Survive()` for each endpoint and repeater using the upset CDF curve. (NOTE: only Series G has upset curves.

8) If the span survives both Steps 6 and 7, then print a '`1`' to the output damage file. Otherwise, print a '`0`'.

9) Tally up the damage stats in `span_dmg_stats[]`.

10) Repeat Steps 6 through 9 for the number of iterations specified by `tot_iter[]` for the current EMP stress level.

11) Repeat Step 10 for each EMP stress level.

Inputs none

Outputs none

Purpose To print summary statistics for each span type.

Called By `main()`

Calls To none

Local Variables

double `mean` average length of spans in a category
`sdev` standard deviation of the length of spans in a category

Global Variables

integer `span_stats[11]` counts the number of spans in each span category
`span_dmg_stats[11][2][3]`
span damage stats by category (out of 11) and damage level (out of 3) where the middle subscript allows for holding live and damage totals

Algorithmic Description

This function prints out span statistics. For each category equipment (e.g., T1), the total number of spans in the category are printed, as well as the percentage of all spans that that category accounts for. The average length of the spans in the category is also printed. Finally, the percentage of spans damaged at each EMP stress level (low, medium, high) is shown.

Inputs		
character	`equip[]`	a 3-character equipment code

Outputs

globals — structure `EQlist` of type `EQ`
with fields:

character	`eq_type[4]`	equipment code
integer	`freq`	number of occurrences
pointer	`ptrnext`	points to next item in `EQ_list`

Purpose To maintain a list of unknown equipment codes and the number of times each code appears..

Called By
`DmgNode()`
`DmgSpan()`

Calls To none

Local Variables

integer	`flag`	an indicator variable
pointer to structures	`EQ_ptr`	points to an `EQ` entry
	`EQ_new`	points to a new `EQ` entry to be inserted into `EQ_list`
	`EQ_prev`	points to an `EQ` entry

Global Variables none

Algorithmic Description

There are hundreds of equipment codes in use in the LECs. Not all of these codes have been assigned to a CDF curve. This routine tallies these "unknown" codes for each data file. Upon being sent an unknown code, this routine checks the existing list of codes (`EQ_list`). If found, then the number of occurrences is incremented. Otherwise, the code is added to the end of the list.

Inputs

integer	`curve`	a CDF curve number

Outputs

returns	integer	returns either TRUE (1) indicating survival or FALSE (0) indicating damage

Purpose To determine equipment survivability from a specified CDF curve.

Called By
`DmgNode()`
`DmgSpan()`

Calls To `detprb()`

Local Variables

integer	`answer`	holds the return value (survival or failure)
double	`prob1`	a random probability from the CDF curve
	`prob2`	a random number (uniform, 0-1)

Global Variables

integer	stream_num2	random number stream #2

Algorithmic Description

This function performs a single "CDF Test" described in the `damage` module. It picks a random probability from a specified CDF curve (`prob1`) and compares it with a random number (`prob2`). If `prob1`≥`prob2` and `prob1` does not equal 0, then Survive returns TRUE (1) indicating equipment survival. Otherwise, it returns FALSE (0) ind

Inputs

integer	`curve`	the number of a CDF curve

Outputs

returns	double	the probability associated with a random point on the input CDF curve

Purpose This routine picks a random point on the input CDF curve and returns the associated probability. For Series G and FD-565 optical fiber equipment, supplemental data points are read from supplemental CDF curves to improve the precision of the curves. The supplemental data is used if the random probability falls between 0.9 and 1.0.

Called By `Survive()`
`DmgSpan()`

Calls To none

Local Variables

integer	`i`	an array index number
	`sup_tog`	flag to indicate that the supplemental CDF's should be used
	`sup_ptr`	the supplemental CDF curve associated with the main CDF curve
double	`point`	the random CDF point
	`prob`	the probability associated with `point`
character	`line[80]`	holds an entire line from the keyfile
	`infile[81]`	temporarily holds the name of an input asset file
	`outfile[81]`	temporarily holds the name of an output damage file
	`type`	a single character holding the type of asset file (N=node, S=span)

Global Variables

integer	`stream_num1`	random number stream #1
double	`cdf_table[MAX_CDF][100]`	100-point CDF curves
	`supp_cdf_table[MAX_SUP][100]`	100-point supplemental CDF curves

Algorithmic Description

For a specified CDF curve (`curve`), this routine returns a random probability based on the following procedure:

1) Return 0.0 if the first curve data point is 1.0 (i.e., curve is all dead).

2) Return 1.0 if the last curve data point is 0.0 (i.e., curve is all alive).

3) Pick a random number, A.

4) Step from right to left through the CDF curve until the y-value of the current data point is less than A.

5) Check if a supplemental CDF curve is necessary.

6) If not, then return the x-value of the final data point divided by 100.

7) If a supplemental CDF is necessary, then step from right to left through the supplemental CDF curve until the y-value of the current data point is less than A.

8) Return the 0.9 plus the x-value of the final data point divided by 1000.

3.12 mklink: Make Link module

Purpose This module assesses the effect of span and node damage on an IEC network's physical transmission paths. Physical damage is translated into lost trunk group capacity in the logical network.

Call Syntax `mklink <filename>`

where `<filename>` specifies the name of the input file containing a list of all other input and output files

example `mklink MCIfiles.fy94`

Input Files

<u>list file</u> This file simply contains the names of the three input files, one output file, and two user-specified parameters to be used by `mklink`. File names are limited by `mklink` to a length of 50 characters.

format

line1:	<combined trunk/path file name>
line2:	<damaged switch file name>
line3:	<damaged span file name>
line4:	<output "qlink" file name>
line5:	direction flag (0 = bi-directional; 1 = one-way)
line6:	damage vector count (integer)

<u>damaged switch file</u> This file contains the list of codes for IEC backbone switches, followed by a user-specified number of damage vectors, where 0 specifies that the switch has been damaged, and 1 specifies that it is functional.

format <IEC switch CLLI code>, <damage vector string of 0/1's>
(c11, 1x, n(i1)), where n is number of damage vectors

example `ADMDTX0101T 11011111001101`

<u>damaged span file</u> This file contains the list of IEC network spans, specified by the span endpoint codes, followed by a user-specified number of damage vectors. Span endpoint codes that are not full 11-character CLLI codes are padded with blanks.

format <endpoint A code>, <endpoint B code>, <damage vector string of 0/1's>
(c11, 1x, c11, 1x, n(i1)), where n is number of damage vectors

example `AKRNOHXX    ALBQNM2505T 11011111001101`

<u>trunk/path file</u> This is the file produced by the `mkpath` module, which details how many trunks traverse each physical path in the IEC network. The file specifies the IEC switch CLLI codes of the trunk/path endpoints, the number of trunks in the A to Z direction (or all bi-directional trunks), the number of trunks in the Z to A direction (only used for one-way

trunk groups), and a series of span index numbers that define the physical transmission path.

format — <switch inaex A>, <switch index Z>, <A->Z trunk quantity>, <Z->A trunk quantity>, <span1>, <span2>, ... , <span n>
(i6, i6, i6, i6, [spans:] i6, i6, ... , i6)

Output Files

<u>output 'qlink' file</u> — This file essentially replaces the path information (string of span indices) from the trunk/path file with a damage vector that indicates whether the path is damaged. In addition, since there is only one trunk size field in each qlink output record, a one-way Z->A trunk group (the second trunk group field in the input trunk/path file) is handled by creating a second qlink record, with the endpoints placed in reverse order. In this sense, the ordering of endpoints in the qlink file may represent the directionality of the trunk group. A record number has been added as the first field.

format — <record #>, <switch index A>, <switch index Z>, <trunk quantity>, <damage vector string of 0/1's>
(i5, i4, i4, i4, 1x, n(i1)), where n is number of damage vectors

example

```
391   22   23   48 11011111001101
392   22   24   96 01110111101011
```

Includes

```
<stdio.h>
<string.h>
<math.h>
"fileio.c"      See Appendix B
```

Constants

MAXLENGTH	4000	maximum number of characters in a path file record
CLLI_LNG	12	length of a switch CLLI code, including terminating null character
MAX_ITER	101	maximum number of switch and span damage vectors allowed
PATHPRINT	9	defines a path record case for which to print debug data
DMGPRINT	84	defines a damage vector case for which to print debug data
MAX_SPAN_REC	9000	maximum number of records in the span file

Global Variables

file — `*pathfile, *switchfile, *spanfile, *filelist, *linkfile`
pointers to the input and output files

integer
- `direction` — indicates use of bi-directional (0) or one-way (1) trunk groups
- `damage_vec` — specifies number of damage vectors in each damaged switch and span file record

character — `span_damage[MAX_SPAN_REC][MAX_ITER]`
holds the damage vectors for each span file record (e.g., `span_damage[390-1][4-1]` specifies the fourth damage vector for span index 390)

structure — `sw[200]` of type `switch_struct`
with fields:

character	`sw[].clli`	used to hold the list of switch CLLI codes
character	`sw[].damage`	used to hold the switch damage vectors

Component Functions

`openfiles()`	opens input and output files
`readswitches()`	reads in the list of switches and switch damage vectors from the damaged switch file
`readspans()`	reads in the list of spans and span damage vectors from the damaged span file
`createlink()`	maps damage to each trunk/path record and generates output file
`closefiles()`	closes input and output files

Function Tree

```
          ┌─openfiles()
          ├─readswitches()
main()────┼─readspans()
          ├─createlink()
          └─closefiles()
```

Algorithmic Description

The physical IEC network is composed of switches and spans (e.g. repeater-to-repeater transmission segments). Damage to switches and spans is represented deterministically as a set of scenarios, or damage vectors, where a value of 0 represents failure of that asset, and a value of 1 represents no damage. The purpose of this module is to determine the effect of damaged switches and spans on the logical IEC network (i.e. point-to-point trunk group sizes). Damage to these individual network components is mapped to an entire physical transmission path (two switch endpoints connected by a series of spans) to determine if the path fails or survives. The logical network capacity is then adjusted for damage based on the number of trunks that traverse the path.

`Mklink` is an important module because it maps physical damage onto the logical network, so that the significant quantity of physical path data does not need to be carried forward in the data flow to subsequent TAMI modules. The output 'qlink' file will contain the pool of IEC network damage scenarios required for the Monte Carlo sampling methodology employed by TAMI.

The module requires two user-specified run-time parameters, scanned in from line 5 and 6 of the input file. The first is a direction flag, which tells the module whether it should look for one-way trunk group quantities (in both the A->Z and Z->A columns of the trunk/path file) or a single bi-directional trunk group quantity from the A->Z column. The second option is the damage vector count. This parameter tells the module how many damage vectors to expect to read from the damaged switch and span files.

The goal of this module is to evaluate each record in the trunk/path file for damage, replacing the long series of span indices with a string of evaluated 0/1 values that indicate whether the path is damaged or functional for each damage vector. To evaluate the effect of the n^{th} damage vector on a path, the following series of lookups is performed:

1) For each path endpoint, look up the n^{th} damage vector in the list of damaged switches. If either switch endpoint is damaged, the entire path is damaged; if not, we must continue evaluating damage in the next step.

2) For each span in the path record, look up the n^{th} damage vector in the damaged span file. If the span is damaged, the entire path is damaged; if not, we must continue evaluating damage for the next span in the path.

3) If both switch endpoints and all of the spans in a path are undamaged, then the path is undamaged; if at least one part of the path is damaged, the entire path is damaged.

The qlink output file format only supports one trunk quantity field; therefore, in the case of one-way trunk groups, a trunk/path record containing both an A->Z and a Z->A trunk quantity will result in two qlink output records, the first with endpoints A and Z, and the second with endpoints Z and A.

The code for this module is straightforward. The `main()` routine passes the `<filename>` argument into `openfiles()`, which opens input and output files and reads in the user-specified directional flag and number of damage vectors. It then calls `readswitches()` and `readspans()` to load the list of switches and corresponding damage vectors, and spans and corresponding damage vectors. `Createlink()` is called next to perform the damage checking algorithm described above. This routine reads in a trunk/path record, evaluates the switch endpoints for damage, and if necessary, evaluates each component span for damage. Results are printed directly to the output file. If one-way trunk groups are being employed, the `createlink()` function will print two qlink output records–one for each direction. `Closefiles()` is called to close all open files before the module terminates.

Inputs

character	`files`	string containing the name of the file that lists five input/output files, a trunk group directionality indicator and a damage vector count	

Outputs

global	file	`*filelist`	points to the file whose name is contained in the `files` string
		`*pathfile`	points to a file that contains information on the physical paths and trunk sizes
		`*switchfile`	points to a file that contains information on damage to the backbone network switches
		`*spanfile`	points to a file that contains information on damage to the backbone network spans
		`*linkfile`	output QTCM link file
	integer	`direction`	equals 1 if uni-directional trunk groups are being used
		`damage_vec`	the number of damage vectors in the switch and span damage files

returns — no formal values are returned

Purpose To open path, switch and span input files and the QTCM-link output file, and to read in the trunk group directionality indicator vector and the damage vector count.

Called By `main()`

Calls To none

Local Variables

character	`tempfile[80]`	this variable is used temporarily to hold the name of the next file to be opened and read in from the list of files in `filelist`

Global Variables none

Algorithmic Description

This function opens the file whose name is stored in the string `files`, setting a filepointer to `filelist`. `Filelist` contains a list of all the input files to be opened in the following order: `pathfile`, `switchfile`, `spanfile`, and `linkfile`. It also contains a trunk group directionality indicator, `direction`, and a damage vector count, `damage_vec`. This function opens each of the input/output files, in the order they are read in from `filelist`.

Errors encountered during any file opening operation result in an error message being printed to the screen, and termination of the module.

Inputs none; operates on global variables

Outputs

global	switch_struct structure	`sw[]`	used to hold each record in the switch file
	with field		
	character	`sw[].clli`	holds switch endpoints read in from the switchfile
		`sw[].damage`	holds damage vectors for the switches

returns no formal values are returned

Purpose To read the list of switch CLLI codes and damage vectors from the switch damage file into the `sw[]` structure, and to compute summary switch survivability statistics.

Called By `main()`

Calls To none

Local Variables

integer	`i, j`	general loop count variables
	`num_nodes`	counts the number of records read in from the switch file
	`num_live`	counts the number of undamaged switches for a given damage vector
	`dam_temp`	temporarily holds the damage vector read in from the `sw[]` structure
	`len`	toggle indicating end-of-file or length of valid record
float	`num_surv`	the switch survivability percentage for the current damage vector
	`min_surv`	minimum percentage of surviving switches
	`max_surv`	maximum percentage of surviving switches
	`tot_surv`	the sum of the values of `num_surv`
character	`line[]`	used to hold a line of input from the switch file

Global Variables

file	`*switchfile`	points to the switch file

Algorithmic Description This function has two distinct sections.

The first section reads the switch file line by line, loads each CLLI code and damage vector into the `sw[]` structure, sets `num_nodes` equal to the number of switches, and prints out `num_nodes` to the screen.

The second section computes a number of switch survivability statistics, including the cases (damage vectors) that result in minimum and maximum switch survivability over all damage vectors

Inputs none; operates on global variables

Outputs

global character `span_damage[]` used to hold the span damage vector from the span file

returns no formal values are returned

Purpose To read the list of span damage vectors from the span damage file, into the `span_damage` array, computing summary survivability statistics in the process.

Called By `main()`

Calls To none

Local Variables

integer	`i, j`	general loop count variables
	`num_spans`	counts the number of records read in from the span file
	`num_live`	counts the number of spans that are not marked as damaged in the `span_damage` array
	`dam_temp`	temporarily holds the value read in from the `span_damage` array
	`len`	boolean toggle indicating end-of-file
float	`num_surv`	percentage of surviving spans for a given damage vector
	`min_surv`	minimum percentage of surviving spans
	`max_surv`	maximum percentage of surviving spans
	`tot_surv`	the sum of the values of `num_surv`, used to calculate average survival percentage over all damage vectors
character	`line[]`	used to hold a line of input from the span file

Global Variables

file	`*spanfile`	points to the span file

Algorithmic Description

This function has two distinct sections

The first section reads the span file line by line, loads each damage vector into the `span_damage` array, sets `num_nodes` equal to the number of spans, and prints `num_nodes` to the screen as a summary statistic.

The second section, computes further summary statistics, including the minimum and maximum survivability for a given damage vector, and the average survivability over all damage vectors. For each damage vector in the `span_damage` array, it parses the vector, character by character, and loads each character into a temporary variable, `dam_temp`. If the span is undamaged, this routine increments the functional span counter, `num_live`

After the number of functional spans has been counted, this routine calculates the span survival percentage (the ratio of functional spans to total spans) and adds this result to a running total, `tot_surv`.

As the function computes span survivability percentages for each damage vector, it keeps track of the minimum and maximum span survivability.

Finally this function prints the minimum and maximum span survivability of a single damage vector, and the average span survivability over all damage vectors.

Inputs	none; operates on global variables
Outputs	
returns	no formal values are returned; outputs are written directly to the output file.
Purpose	To map damage to each trunk/path record, replacing the long series of span indices with a string of evaluated 0/1 values that indicate whether the path is damaged or functional for each damage vector. and to generate an output file.
Called By	`main()`
Calls To	none

Local Variables

integer	`i, j`	general loop count variables
	`length`	the string length of a `pathfile` line
	`trk1`	the number of trunks in the A ->Z direction
	`trk2`	the number of trunks in the Z ->A direction
	`sw1`	the originating path endpoint
	`sw2`	the terminating path endpoint
	`dmg1`	originating switch endpoint damage value, 0/1
	`dmg2`	terminating switch endpoint damage value, 0/1
	`num_spans`	the number of spans in a path record
	`min_spans`	the number of spans in the shortest path in the pathfile
	`max_spans`	the number of spans in the longest path in the pathfile
	`tot_spans`	the sum of the values of `num_spans` for all paths
	`dead`	toggle for value of switch endpoint damage, 0/1
	`count`	the current record number of the output file, `linkfile`
	`num_path`	the current record number for the input trunk/path file
	`loop`	loop count variable for reading variable number of spans for each path
character	`line[]`	temporarily holds a line of input from the trunk/path file
	`directline[]`	temporarily holds a line of output for the qlink file
	`tmp1[]`	temporarily holds originating switch endpoint damage value , 0/1
	`tmp2[]`	temporarily holds terminating switch endpoint damage value, 0/1
	`span[]`	temporarily holds span damage vector

Global Variables

integer	`direction`	indicates use of bi-directional (0) or one-way (1) trunk groups
	`damage_vec`	specifies number of damage vectors in each damaged switch and span file record

Algorithmic Description

This function processes the trunk/path file record-by-record, loading the path endpoints and the number of trunks.

In order to evaluate the effect of the n^{th} damage vector on a path, the following algorithm is executed: for each path endpoint, this function, looks up the n^{th} damage vector in the list of damaged switches. If either switch endpoint is damaged, the entire path is damaged; if not, the function evaluates each span in the path record. For each span, this function , look up the n^{th} damage vector in the damaged span file. If the span is damaged, the entire path is damaged; if not, this function evaluates the next span in the path for damage. If both switch endpoints and all of the spans in a path are undamaged, then the path is undamaged; if at least one part of the path is damaged, the entire path is damaged. Results are printed directly to the `linkfile` output file. If one-way trunk groups are being employed, this function will print two qlink output records–one for each direction.

Inputs	none		
Outputs			
global	file	`*pathfile`	points to the file that contains combined the physical paths and trunk sizes between switch pairs
		`*switchfile`	points to file that describes damage to the backbone network switches
		`*spanfile`	points to a file that describes damage to the backbone network spans
		`*linkfile`	output QTCM link file
returns	no formal values are returned		

Purpose To close path, switch and span input files and the QTCM link output file.

Called By `main()`

Calls To none

Local Variables none

Global Variables none

Algorithmic Description This function closes the files whose names are pointed to by the following pointers: `pathfile`, `switchfile`, `spanfile` and `linkfile`, in the order given.

Appendix A: ICF File Format Descriptions

The OMNCS maintains information regarding the IEC networks in a format based on an indexed chained format (ICF). The TAMI model requires this data to be converted into a format for use in TAMI analysis. Four of the modules concern themselves with re-arranging the ICF data files into TAMI data files: `span_make`, `array_make`, `mk_ncam_path` and `array_make`. This TAMI data structure more readily lends itself to the type of processing performed in TAMI. The following is a brief discussion of the ICF format.

ICF NETWORK DATA FILE FORMAT

The purpose of this file format is to specify a structure that ensures a common data input for various network simulation models. The file format is based on an indexed chained format (ICF). All raw network data will be converted into ICF. The ICF consists of data files cross indexed in order to provide fast disk access. This format allows coherent logical subsets of the network data to be quickly and easily loaded into simulation models. Thus, every model's data input will be standardized.

The ICF representation of each network consists of four files: node data file, link data file, CG data file, and a path data file. The damage and routing files for each network will not be in ICF.

Each file has a header record which identifies the network and the ICF file. The headers have the format XXX <File> where XXX is the network (FPS for FPSC, FCA for FCAP, MCI for MCI, and SPR for Sprint). <File> can be "link", "path", "trnk" and "node" for the link, pid, cg or node files, respectively. The length of the header record is the same length as the other records in that file.

The following sections will detail each file with an example.

NODE DATA FILE:

- Sorted by node index and CLLI
- All fields are left justified.
- All records end with a carriage return.

The node data file contains the assets of the network.

Node idx	Clli code	Link Head	Link Tail	V Cd.	H Cd.	Extra (Reserved)
I4	C11	I4	I4	I4	I4	C33
123	BLTMD023	11	12	1234	4567	
125	CHCIL009	13	30	3343	1233	
134	KANM0008	31	31	2334	2445	

Total record size = 65 bytes

LINK DATA FILE:

Sorted by link idx
All fields are left justified.
All records end with a carriage return.

The link data file contains the physical connections in the network. The type of link is also represented. The link head from the node data file points to the first record in a block of records. The link tail references the last block.

In this example, node 123 (BLTMD023) is connected to 125 (123 ↔ 125); 123 ↔ 456; 123 ↔ 23; 123 ↔ 654; 123 ↔ 230.

Link idx	Node Idx	Type Con.	Node Idx	Type Con.	Node Idx	Type Con.	Node Idx	Type Con.
I4	I4	C1	I4	C1	I4	C1	I4	C1
11	125	D	456	D	23	G	654	R
12	230	D						
13	4566	Y	223	T	211	T	1211	T
14	32	N	211	W	1111	W	112	W
.								
.								

Total record size = 25 bytes

Link Types

blank	Undefined	
C	(MCI)	Cable
D	Digital	T-Carrier
E	Digital	Zero Loss trunks
G	Analog	Zero Loss trunks
I	Analog	Satellite
L	Analog	L-Carrier L3, L4, L5
N	Analog	N-Carrier
P	Undefined	Assumed inter-building link (used in AT&T data)
R	Analog	Radio systems
T	Analog	Coaxial Systems T4
U	Undefined	Hybrids
V	Digital	Generic Future Digital Technology
W	Digital	Fiber Optic
Y	Digital	Radio Systems
Z	Digital	Leased Digital Capacity

CG DATA FILE: Sorted by CG idx
All fields are left justified.
All records end with a carriage return.

The CG data file contains the logical connections of the network. The path head points to the first record in a block of records in the path data file. The path tail references the last block. These records detail the physical paths that comprise the CG. Node A and node Z reference the node data file, which are the node end points of the trunk group. The TRK qty specifies the number of trunks in a trunk group. The type identifies the grade of service of the CG, and the dir field specifies the direction of the CG trunk.

CG Idx	Path Head	Path Tail	Node A Idx	Node Z Idx	TRK Qty	Dir	Type	Extra (Reserved)
I5	I6	I6	I4	I4	I4	C1	C2	C6
1	32	33	123	125	23	B	DN	
2	34	37	125	140	21	A	AF	
3	38	34	125	140	12	Z	PH	
.								
.								

Total record size = 39 bytes

CG Types:

IT	Intermachine Trunk
PH	Primary High Usage
AF	Alternate Final
DN	Dynamic Nonhierarchial

DIRECTION:

B	Bi-directional
A	From A to Z
Z	From Z to A
blank	Bi-directional

PATH DATA FILE:

- Sorted by Path idx
- All fields are left justified.
- All records end with a carriage return.

The path data file contains the paths and the nodes of a CG. For example CG idx 1 contains only one path, pid no. 1 and it is comprised of the nodes:

123 ↔ 134 ↔ 231 ↔ 12 ↔ 16 ↔ 24 ↔ 46 ↔ 25 ↔ 55 ↔ 42 ↔ 223 ↔ 456 ↔ 125.

CG idx 2 has three paths, pid nos. 2, 3, and 4.

CG idx 3 has one path, pid no. 5.

Path Idx	Pid No.	CG Idx	Node Idx	Node Idx	Node Idx	Node Idx	Node Idx	Node Idx	Node Idx
I6	I5	I5	I4	I4	I4	I4	I4	I4	I4
32	1	1	123	134	231	12	16	24	46
33	1	1	25	55	42	223	456	125	
34	2	2	125	140					
35	3	2	125	156	312	312	1234	123	346
36	3	2	123	140					
37	4	2	125	234	140				
38	5	3	125	260	140				

Total record size = 45 bytes

Appendix B: User-Defined Utility Functions

Functions that are repeatedly utilized by more than one module have been placed in this appendix in order to make them readily available. These "utility" functions are divided into two groups:

1) Function calls repeatedly coded into various modules

```
fget()
char_comp()
```

2) Function calls included in include "fileio.c":

```
parse()
parse_int()
getline()
fopenfile()
```

fget function

Inputs

integer	`fp`	a pointer to a file, equivalent to type `FILE`
	`num`	indicates the number of bytes to read from the current position
long integer	`pos`	indicates a position within the file pointed to by `fp`
character	`data`	the buffer to hold data read from the file

Outputs

returns	integer	returns the number of bytes read, or `-1` if error

Purpose This function reads a specified number of characters into a string buffer from a given position within the input file.

Local Variables none

Global Variables none

Algorithmic Description This utility I/O function uses the `<stdio.h>` function, `fseek()`, to set the file pointer `fp` to position `pos`, the position of the first byte to be read. The function then uses `fread()` to read `num` bytes into buffer string `data`. If there is an error, a value of `-1` is returned; otherwise, the return value specifies the number of bytes read.

Inputs

character	*cmp1	points to the first string passed into `char_comp()` for comparison
	*cmp2	points to the second string passed into `char_comp()` for comparison

Outputs

returns	integer	this function returns a 0 if the two strings are equal, and returns a non-zero if they are different

Purpose To compare two character strings, for use in sorting `(qsort())` and searching `(bsearch())`

Local Variables none

Global Variables none

Algorithmic Description This function is used by `bsearch()` and `qsort()` to compare two strings. The arguments `cmp1, cmp2` are passed into the standard 'C' `strcmp` function, and the result is used as the `char_comp()`'s return value. The result is 0 if `cmp1=cmp2`, and non-zero otherwise.

file *fopenfile(filename, type) **fileio.c**

This utility function is a simple modification of the standard 'C' `fopen` function. It opens the passed in filename and checks for an error in the file. If an error exists the function is exited.

void parse(start, num, buffer, rtn) **fileio.c**

This utility function reads `num` characters from the input character string `buffer` starting at position `start` and directs the output to the character string `rtn`.

int parse_int(start, num, buffer) **fileio.c**

This utility function reads `num` characters from the input character string `buffer` starting at position `start` and returns the integer value of the characters

int getline(fildes, buf) **fileio.c**

This utility function reads from the file `fildes` until the first end-of-line character is reached, and directs the output to the buffer `buf`

List of Acronyms

AT&T	American Telephone & Telegraph
CSF	Cumulative Distribution Function
IEC	Inter-Exchange Carrier
ICF	Indexed Chain Format
LEC	Local Exchange Carrier
MCI	MCI Telecommunication Corporation
NCAM	Network Connectivity Analysis Model
NCS	National Communication System
NLP	National Level NS/EP Telecommunications Program
NS/EP	National Security and Emergency Preparedness
NT	National Communications System (OMNCS) Office of Technology and Standards
OMNCS	Office of the Manager, National Communication System
PSN	Public Switched Network
QTCM	Queuing Traffic Congestion Model
TAMI	Traffic Analysis by Method of Iteration
TG	Trunk Group

List of References

1. QTCM Software Documentation, Volume I: Programmer's Manual, National Communications System, November 1990.

2. Network Analysis Sensitivity Report, National Communications System, March 1994.

3. Network Analysis Report, National Communications System, June 1994.

4. Infrastructure Damage Assessment/Communication Assessment Model, Programmer's Manual, National Communications System, October 1990.

5. Network-Level EMP Effects Evaluation On The Primary PSN Toll-Level Networks, National Communications System, June 1993.

6. Network Congestion Analysis Report, National Communications System, November 1992.