

# CHGEN User's Manual Version 10log

Computer Science Department  
University of Massachusetts at Lowell

Chien-Yuan Wang  
Ching-Chih Ko  
Ke-Chiang Chen  
Chih-Yung Wu

Revised for genlog by  
Mike Murphy  
Denise Nelson  
Bill Rideout  
in 96s523

May 14, 1996

# Contents

<b>1</b>	<b>Overview of CHGEN</b>	<b>1</b>
1.1	The Data Model . . . . .	1
<b>2</b>	<b>Using CHGEN</b>	<b>2</b>
2.1	Running CHGEN . . . . .	2
2.2	Environment . . . . .	2
2.3	Schema Format . . . . .	2
2.3.1	CHGEN 10 schema . . . . .	2
2.3.2	GENDB v1.0 schema . . . . .	6
2.3.3	Sample Schema Files . . . . .	7
2.4	parts.sch . . . . .	7
2.5	college.sch . . . . .	9
2.6	Output . . . . .	11
2.6.1	metaschema.dat . . . . .	11
2.6.2	pr utilities . . . . .	12
2.7	Limitations . . . . .	12
2.8	Error Conditions . . . . .	13
2.9	Command Line Switches . . . . .	15
<b>3</b>	<b>Writing Applications</b>	<b>17</b>
3.1	Include Files . . . . .	17
3.2	”C” Files . . . . .	17
3.3	Global Variables . . . . .	17
<b>4</b>	<b>Datafile Formats</b>	<b>18</b>
4.1	General Format . . . . .	18
4.2	String Fields . . . . .	19
<b>5</b>	<b>Using the pr_ Routines</b>	<b>19</b>
5.1	pr_init . . . . .	19
5.2	pr_load . . . . .	20
5.3	pr_create . . . . .	21

5.4	pr_add . . . . .	21
5.5	pr_dump . . . . .	22
5.6	pr_dump_row . . . . .	23
5.7	pr_free . . . . .	24
5.8	pr_delete . . . . .	25
5.9	pr_find . . . . .	27
5.10	pr_rcount . . . . .	28
5.11	pr_stats . . . . .	28
5.12	pr_startlog . . . . .	29
5.13	pr_stoplog . . . . .	30
5.14	pr_replay . . . . .	30
<b>6</b>	<b>Key Manipulation Functions</b>	<b>31</b>
6.1	decode . . . . .	31
6.2	decode_retstr . . . . .	32
6.3	encode . . . . .	33
6.4	key_compare . . . . .	33
<b>7</b>	<b>Iterators and Other Macros</b>	<b>34</b>
7.1	table_loop . . . . .	34
7.2	child_loop . . . . .	35
7.3	first_child . . . . .	35
7.4	last_child . . . . .	36
7.5	find_str_loop . . . . .	37
7.6	find_key_loop . . . . .	38
7.7	find_int_loop . . . . .	38
7.8	pr_set_int . . . . .	39
7.9	pr_set_ft . . . . .	39
7.10	pr_set_str . . . . .	40
7.11	pr_get_int . . . . .	41
7.12	pr_get_ft . . . . .	41
7.13	pr_get_str . . . . .	42

7.14	pr_set_default . . . . .	42
7.15	pr_get_default . . . . .	43
7.16	pr_set_key . . . . .	43
7.17	pr_get_key . . . . .	43
7.18	" Access Macros" . . . . .	44
7.19	User-Variable Macros . . . . .	46
<b>8</b>	<b>View Definition Files</b>	<b>46</b>
<b>9</b>	<b>A Complete Example</b>	<b>47</b>
<b>10</b>	<b>Problems Running Version 10</b>	<b>58</b>
10.1	CHGEN-F-NAMETOOLONG . . . . .	58
10.2	CHGEN-F-MISSINGTBLSTART . . . . .	58
10.3	CHGEN-F-NOPATH . . . . .	58
10.4	CHGEN-F-ERRNOPARENT . . . . .	59
10.5	CHGEN-F-DUPTBLID . . . . .	59
10.6	No metaschema.dat . . . . .	59
<b>11</b>	<b>Changes Made in Version 10-log</b>	<b>59</b>
<b>12</b>	<b>Changes Made in Version 10</b>	<b>60</b>
<b>13</b>	<b>Release Notes 9.1</b>	<b>60</b>
<b>14</b>	<b>Release Notes 8.0</b>	<b>61</b>
14.1	Running CHGEN . . . . .	61
14.2	Data File Changes . . . . .	62
14.3	Programming Interface . . . . .	62
<b>15</b>	<b>Release Notes (from version 6.0)</b>	<b>64</b>
15.1	Running CHGEN . . . . .	64
15.2	Data File Changes . . . . .	64
15.3	Programming Interface . . . . .	64
15.4	Miscellaneous . . . . .	66

<b>16 CHGEN 10log</b>	<b>66</b>
16.1 Where to find CHGEN v10log . . . . .	66
16.2 Running CHGEN 10log . . . . .	66
16.3 Output of CHGEN 10log . . . . .	66
<b>17 CHGEN 10</b>	<b>66</b>
17.1 Where to find CHGEN v9.1 . . . . .	66
17.2 Running CHGEN 10 . . . . .	67
17.3 Output of CHGEN 10 . . . . .	67
<b>18 Appendices</b>	<b>67</b>
18.1 Appendix A: entry-point file cross-reference . . . . .	67
<b>19 References</b>	<b>69</b>

# 1 Overview of CHGEN

This file is in /usr/proj3/case/gen/ver\_10log/doc/genv10log\_manual.\*.

The name CHGEN stands for “C” and “H” generator. The name is appropriate since CHGEN outputs a set of code in the C programming language with the names ending in .c and .h. It was developed over five years time at the University of Massachusetts, Lowell formerly the University of Lowell. T.C. Cheng and C.Y. Chou created the initial version during the summer of 1990. S.C. Smith and C.E. Smith continued development during the spring and summer of 1991.

The purpose of CHGEN is to aid programmers in developing database code for an ongoing CASE tool project within the Computer Science Department’s Software Engineering course sequence. Its extensible data model minimizes the impact of changing requirements during evolutionary development of complex software projects. Its graphic symbols and standard naming conventions improve communication and documentation effectiveness. It increases programmer productivity by providing internal network data structures, access macros, and import/export utilities for single-user computations on complex objects. It also supports version management and persistent storage in external files, and shared access in an INGRES relational database.

## 1.1 The Data Model

CHGEN is a code generation tool which supports a data model-based approach to programming. The data model used is that of a network database, where tables are arranged in a parent-child graph. Relation links are implicitly added to each table’s explicit definition in the form of pointer chains. These allow programmers to freely traverse records of related data in the tables. The model enforces that when relationships are drawn as link between nodes (tables), parents are above and children are below.

The cardinality of the relation can be 1 to 1, or 1 to many, but never many to many. Relations with many to many cardinality are common in relational database design, but can be easily replaced by two 1 to many relations plus a new associative table. The associative entity provides a placeholder for the relation’s attributes. In addition, the parent side can never be the “many” side. The relation is explicitly defined in the associative or child table.

Finally, each record in a table is assigned a unique object identifier, called the primary key (abbreviated as pkey). The pkey is an eight character ASCII string. The first two letters is the table abbreviation. The next two digits are the version of the dataset that the record belongs to. The final four digits are the record, or sequence number of the record within the table and version. For example, the pkey “AA030042” belongs to record number 42 in version 3 of the data within the “AA” table. Do not assume that the record number implies that the records are stored in an array. In fact, a doubly linked presorted list is used. The number simply comes from an ever increasing counter used to assign record numbers when records are created (within each table/version set).

## 2 Using CHGEN

### 2.1 Running CHGEN

<sup>1</sup>/

You can find the current executable file to run CHGEN in the directory:

```
/usr/proj3/case/95s523/genv10/base/executables/chgen10
```

If you can not find the executable file to run the current version of CHGEN, you must copy the source files into your own directory and build one by running Makefile ( under prompt "%", type command "make" to make "chgen10" this execution file ). The source files can be found in:

```
/usr/proj3/case/95s523/genv10/base/src/chgen10
```

Once you have created the executable, you no longer need the source files.

To run the program:

```
chgen10 [command line switches if any] schema_file_name.sch
```

You can leave off the ".sch" suffix when entering the schema file name. The program will append it at runtime.

### 2.2 Environment

This version of CHGEN does not use graphics and can be run from any VT100 terminal. In the UMass Lowell computer science department as of Spring 1995, you can run this program on the Dungeon terminals, dial up modem terminals, or the workstations.

### 2.3 Schema Format

There is no great different in using a CHGEN 10 schema file from CHGEN 9.0. The same command line switches are used and the same utility files are generated. Except that the format of schema file for CHGEN 10 is a little different from that of CHGEN 9.0. And the only difference is on the third table declaration line (the comment field).

#### 2.3.1 CHGEN 10 schema

Example CHGEN 10 schema file (school.sch)

```
students AA /*1 Table of student records */
```

---

<sup>1</sup>[This section is about genv10, not genlog. Genlog was moved to directory /usr/proj3/case/gen\_ver\_10log - RJL]

```

{
  AAid  students_id  c8  1  /* primary key field */
  fname  fname      c30 0  /* first name of student */
  lname  lname      c30 0  /* last name of student */
}

roster BB /*1      Table associating students with courses */
{
  BBid  roster_id   c8  1  /* primary key field */
  AAid  student_id  c8  1  /* foreign key field */
  cname  cname      c30 0  /* name of course */
}

```

This example declares two tables. The first table is named “students” and the second table is named “roster”. The abbreviation for the first table is “AA” and that for the second table is “BB”. The abbreviations must be two characters long and unique. This is very important since these two letters are used as the identifier to the table. After the abbreviations, CHGEN 10 will check the third character of table’s comment line ( made in CHGEN 10), if it is set to ”1”, then it will search by binary tree. Otherwise, it is default as CHGEN 9.0 (sequential search).

Listed as components of each table is a set of attributes. In this example, each table has three. Each attribute line has five items describing it.

The first item in the line is the name of the field. This name must be unique within the table.

The second item is the alternate name, or language-specific type name and is currently not used in this version of CHGEN.

The third item describes the type of the field. The legal types are listed below:

**i2,i4** This is an integer type as defined in the C language.

**cX** This is a character string of width “X”. If the data within an element uses fewer characters than the number specified in “X”, the string is padded with spaces. This type can not have embedded whitespace.

**f4,f8** This is a C language floating point number.

**tX** This is a text character string type. This is the same as cX but strings are not padded with spaces.

The fourth item indicates if the item is a data field or an fkey.

**0** A 0 in this position indicates that this is a data field.

**1** A 1 in this position indicates that this is either a primary key or a foreign key. A primary key must appear as the first attribute in a table’s attribute list.

**-1** A -1 in this position indicates that this item is a foreign key and that back pointers along this relationship should not be generated. If the application you are using will not need back pointers along this chain, this will reduce the memory and code requirements.

s An s in this position indicates that this is a foreign key and the relationship is a singleton (1 to 1). This will also reduce code and memory requirements.

Remember that the primary key appears first in the list of attributes and the foreign keys follow it before the data fields appear in the list of attributes.

The fifth and final item is a comment line. This is not necessary but a warning will be generated for each line that does not have a comment. These comments will be preserved in metaschema.dat which will be discussed later.

Note that in table BB, table AA is referenced as a foreign key. The name specified for this attribute is AAid. This is legal since it begins with the two letter abbreviation for the table (AA) and the entire name is unique within the attributes of table BB.

CHGEN expects the input schema file to follow a specific format. The schema file is composed of 'blocks', where each block defines a single database table. The following template demonstrates the supported schema format:

```
        tablename tableabbrev /*1 TT comment line */
    {
pkey  primary_key  c8  1 /* TA comment line */
fkey1  foreign_key1  c8  1
    o
fkeyN  foreign_keyN  c8  1
field1  data_field1  type  0
    o
fieldN  data_fieldN  type  0
    }
```

In this template, the following definitions and restrictions hold:

**tablename** The name of the table as stored in Ingres, MUST be 12 characters or less, can only contain lowercase letters and digits, and cannot start with a digit.

**tableabbrev** MUST be 2 or 4 uppercase characters, and MUST be unique amongst all other tableabbrev's in the schema, since this is the name of the table in the network data-structures.

**TT comment line** If you want to do binary search on pkey only, set the third character to "1". Otherwise, CHGEN 10 will use sequential search.

For example, the following is valid:

CHGEN 9.0:

```
tableattrs TT /* TT: - attribute (field) definitions... */
```

CHGEN 10:

```
tableattrs TT /*1 It will use binary search... */
```

```
tableattrs TT /* 1 It will use sequential search (for "1" is not the third character*/
```

```
tableattrs TT /*0 It will use sequential search... */
```

```
tableattrs TT ( NOTE: Without comment field, it still use sequential search)
```

{ Marks the start of the list of fields within the table.

**pkey** MUST start with the tableabbrev, followed by the letters 'id'. For example, TTid is a valid primarykey for the TT table, but ttid, TT\_id, TTid2, TT\_key, etc... are not valid. To conform with the relational database model being followed, all key fields MUST be of type c8 (8 character string) regardless of external size (8 or 12 characters). The "1" digit appearing in the line indicates that this field is a key field. Every table MUST have a primary key, and it MUST be the first field.

**primary\_key** The name of the primary key field in Ingres.

**type** Each field MUST have a data type. The valid types are:

**i2,i4** Integer ("C" type int) NOTE: Contrary to popular belief, types such as i3, i5, etc... are NOT VALID! This misunderstanding may stem from Ingres' VIFRED (VIsual FoRms EDitor) notation, in which i3 implies an integer that is displayed on the form as 3 digits.

**cX** Character string of width "X" ("C" type char fieldname[X+1]). if the data within a cX field does not fill the width of the field, it is automatically padded with spaces. This fact should be taken into account when writing applications that use strcmp.

**f4,f8** Floating point number ("C" type float).

**date** Ingres date type ("C" type char fieldname[26]).

**tX** Text character string type. This is exactly the same as the cX type except that short strings are not padded with spaces.

**field1..N** These fields represent the actual data fields of the table. A Table can have any number of data fields (including zero). Each field name follows the same rules as the table name (12 characters or less, letters and digits, and cannot start with a digit). The field1..N label is the name of the field in the network data-structures, and the datafield1..N label is the field name in Ingres. These fields MUST have a "0" in the 'is key?' field.

**fkey1..N** A table requires one 'foreign key' to each parent table it has. A table can have any number of foreign keys. The actual name of the fkey MUST be either the name of the parents pkey, OR the parents pkey, followed by a digit, followed by letters (if desired). The digits should only be used if the same table is listed as a parent multiple times. For example, if the table FL has the TD table as a 'multiple parent', then some possible values of the two fkeys are :

- TDid, TDid2
- TDid1, TDid2
- TDid1\_stype, TDid2\_field

Internally, this restriction exists because CHGEN needs to locate which table the foreign key is referencing, and assumes either the entire fkey name is the parents primary key, or it recognizes everything before a digit as being a parents primary key. The labels foreign\_key1..N specify the names of the foreign key fields in Ingres. All foreign keys MUST be explicitly marked as keys (1, -1 or s), and MUST be of type c8. Finally, a table can have any number (including zero) foreign keys. The special value "-1" in the is-key field indicates that back-pointers along this relationship chain SHOULD NOT be generated. If your application(s) will never need to traverse backwards along this chain, you should not use back pointers. In this

way, the memory and code requirements of your application(s) may be considerably reduced. The value "s" means that this relationship is 1-to-1 (called a singleton relationship). This is always the case for "is-a" relationships. If you know that a relationship is 1-to-1, you should use the "s" syntax to achieve further code and memory reductions.

} Marks the end of the tables definition.

Finally, comments ("C" language style) can appear almost anywhere in the file, but MUST be either the last, or only item, in a line. This is because CHGEN attempts to locate the 'start comment' symbol in a line ( /\* ). If found, the remainder of the line is considered to be the comment. Inside the comment, CHGEN 10 will check the character of table's, If it is set to "1", then it will generate by binary tree index on pkey.

For example, the following is valid :

```
tableattrs TA /* TA: - attribute (field) definitions... */
tableattrs TA /*1 TA: - attribute (field) definitions... */
tableattrs TA /*0 TA: - attribute (field) definitions... */
```

But this is NOT valid:

```
tableattrs /* TA: - attribute (field) definitions... */ TA
```

NOTE: It is important that you comment your schema files, since the comment is the only way to indicate the purpose of the tables and the fields within them (ie, data definition). Although CHGEN only issues warning messages for missing comments, the db\_load/unload routines will generate actual errors for missing ttabbrev.

### 2.3.2 GENDB v1.0 schema

The format of the schema file used in the GENDB application is slightly different. The most important thing to note is that there is no primary key field specified in the attributes section of any table. CHGEN v9.0 uses the same routines used in version 8.0 so the elements not specified in the GENDB schema are generated internally.

Example GENDB 1.0 schema file (schooldb.sch)

```
create table students AA /* Table of student records */
{
  fname      c(30)    /* first name of student */
  lname      c(30)    /* last name of student */
}

create table roster BB /* Table associating students with courses */
{
  member     parent(students,student_id,1,100) /* foreign key field */
  cname      c(30)   /* name of course */
}
```

This GENDB schema file is the functional equivalent of the previous CHGEN v8.0 schema file. Note that the general format is more compact than the previous schema format.

A table is defined with the words “create table” and then the name of the table and the two letter abbreviation follows. A comment can also follow the table abbreviation like the previous schema format.

The attributes are listed slightly differently. The primary key is now gone and foreign keys are defined in a much different manner. The data fields are also more compact. Let’s examine the foreign key definitions.

In the example, the word “member” is a relation and not a data type. The fact that this is a foreign key is established by the “parent()” declaration. The first field in parent() is the name of the table that is being referenced. In this case, the table’s name is “students”. The second field is the name of the relation. This does not have to be different from the table name. The next two fields are not used in this version of CHGEN. In GENDB, the third value represents the cardinality at the parent’s side and must be 0 or 1. The fourth value represents the cardinality at the child side and can be any non negative number. In this example, the 100 means that up to 100 children can exist for each parent. GENDB does not enforce cardinality at this time but these values are parsed for correctness.

The data fields are similar to those in the CHGEN schema format. The name must be unique within the attributes of the table and the data type must be one of those described above. The only exception is that character string types must be defined in parenthesis as shown in the example. The parenthesis are not needed for any other data type.

GENDB v1.0 operates independently of any version of CHGEN. It is important to note that CHGEN v9.0 runs with the same routines used by CHGEN v8.0 and outputs the same utility files. Despite the fact that version 9.0 accepts schema formats similar to GENDB, the program behaves in the same manner as CHGEN. The “path” mechanism supported by GENDB is not supported by CHGEN so some legal GENDB schema files will not work with CHGEN.

### 2.3.3 Sample Schema Files

The following sample schema files are provided as examples of syntactically correct schema files. They are not necessarily useful for any purpose.

## 2.4 parts.sch

CHGEN v8.0 style schema “parts.sch”

```
table_part PT /* Individual Part */
{
  PTid      part_id      c8 1 /* primary key field */
  part_id   pid          c7 0 /* part id */
  loc_id    lid          c3 0 /* location */
  weight    weigh       f8 0 /* weight */
}

apartoper AP /* Part Operator */
{
```

```

APid      apartoper_id  c8  1 /* primary key field */
PTid      oper_list    c8  1 /* fkey to part */
oper_num  onum         c3  0 /* operator number */
defect_count dcoun      i4  0 /* number of defects */
start     start        c20 0 /* start point */
stop      stop         c20 0 /* stop point */
}

measchar MC /* Measurements */
{
  MCid     measchar_id   c8  1 /* primary key field */
  APid     cut_measurements c8  1 /* fkey to part oper */
  AP02     measurements  c8  1 /* fkey to part oper */
  meas     meas          f8  0 /* measurement */
  person   perso        c20 0 /* person */
}

```

GENDB v1.0 style schema "parts.sch"

```

create table part PT /* Individual Part */
{
  part_id c(7) /* part id */
  loc_id c(3) /* location */
  weight f /* weight */
}

create table apartoper AP /* Part Operator */
{
  part parent(part,oper_list,1,20) /* fkey to part */
  oper_num c(3) /* operator number */
  defect_count i /* number of defects */
  start c(20) /* start point */
  stop c(20) /* stop point */
}

create table measchar MC /* Measurements */
{
  cut_oper parent(apartoper,cut_measurements,1,100) /* fkey to part oper */
  meas_oper parent(apartoper,measurements,1,100) /* fkey to part oper */
  meas f /* measurement */
  person c(20) /* person */
}

```

"mateschema.dat" for "parts.sch" (gendb)

SV010001 PJ010001 parts.sch GENDB CHGEN 9.0

```

TT010001 SV010001 PT part /* Individual Part */
TT010002 SV010001 AP apartoper /* Part Operator */
TT010003 SV010001 MC measchar /* Measurements */

TA010001 TT010001 PTid part_id c8 1 /* primary key field */
TA010002 TT010001 part_id pid c7 0 /* part id */
TA010003 TT010001 loc_cd lcd c3 0 /* location */
TA010004 TT010001 weight weigh f8 0 /* weight */

TA010005 TT010002 APid apartoper_id c8 1 /* primary key field */
TA010006 TT010002 PTid oper_list c8 1 /* fkey to part */
TA010007 TT010002 oper_num onum c3 0 /* operator number */
TA010008 TT010002 defect_count dcoun i4 0 /* number of defects */
TA010009 TT010002 start start c20 0 /* start point */
TA010010 TT010002 stop stop c20 0 /* stop point */

TA010011 TT010003 MCid measchar_id c8 1 /* primary key field */
TA010012 TT010003 APid cut_measurements c8 1 /* fkey to part oper */
TA010013 TT010003 AP02 measurement c8 1 /* fkey to part oper */
TA010014 TT010003 meas meas f8 0 /* measurement */
TA010015 TT010003 person perso c20 0 /* person */

```

## 2.5 college.sch

CHGEN v8.0 style schema "college.sch"

```

teacher CC /* table CC associates courses with professors */
{
  CCid teacher_id  c8  1 /* primary key field */
  tname tname     c30  0 /* teacher's name */
}
room DD /* rooms for courses */
{
  DDid room_id    c8  1 /* primary key field */
  room room      i4  0 /* room number */
}
roster BB /* table BB associates courses with teachers & rooms */
{
  BBid roster_id  c8      1 /* primary key field */
  CCid teacher_id c8      s /* one teacher per course */
  DDid where_id   c8      s /* one room per course */
  cname cname    c30  0 /* catalog name of course */
}
students AA /* table AA holds student records */
{
  AAid students_id  c8      1 /* primary key field */
  fname first_name c30  0 /* first name of student */
}

```

```

lname last_name c30 0 /* last name of student */
}

enrollment EE /* enrollment relation */
{
  EEid enrollment_id  c8  1 /* primary key field */
  AAid student        c8  1 /* fkey to student */
  BBid course         c8  1 /* fkey to course */
}

```

GENDB v1.0 style schema "college.sch"

```

create table teacher CC /* table CC associates courses with professors */
{
  tname          c(30) /* teacher's name */
}

```

```

create table room DD /* rooms for courses */
{
  room i        /* room number */
}

```

```

create table roster BB /* table BB associates courses with teachers & rooms */
{
  courseteach  parent(teacher,teacher_id,1,100) /* one teacher per course */
  courseroom   parent(room,where_id,1,100) /* one room per course */
  cname c(30) /* catalog name of course */
}

```

```

create table students AA
{
  fname      c(30) /* first name of student */
  lname      c(30) /* last name of student */
}

```

```

create table enrollment EE /* enrollment relation */
{
  member  parent(students,student,1,100) /* fkey to student */
  course  parent(roster,course,1,100) /* fkey to course */
}

```

"metaschema.dat" for "college.sch" CHGEN style

SV010001 PJ010001 college.sch CHGEN CHGEN 9.0

```

TT010001 SV010001 CC teacher /* table CC associates courses with professors */
TT010002 SV010001 DD room /* rooms for courses */

```

```

TT010003 SV010001 BB roster /* table BB associates courses with teachers & rooms */
TT010004 SV010001 AA students /* table AA holds student records */
TT010005 SV010001 EE enrollment /* enrollment relation */

TA010001 TT010001 CCid teacher_id c8 1 /* primary key field */
TA010002 TT010001 tname tname c30 0 /* teacher's name */

TA010003 TT010002 DDid room_id c8 1 /* primary key field */
TA010004 TT010002 room room i4 0 /* room number */

TA010005 TT010003 BBid roster_id c8 1 /* primary key field */
TA010006 TT010003 CCid teacher_id c8 s /* one teacher per course */
TA010007 TT010003 DDid where_id c8 s /* one room per course */
TA010008 TT010003 cname cname c30 0 /* catalog name of course */

TA010009 TT010004 AAid stidents_id c8 1 /* primary key field */
TA010010 TT010004 fname first_name c30 0 /* first name of student */
TA010011 TT010004 lname last_name c30 0 /* last name of student */

TA010012 TT010005 EEid enrollment_id c8 1 /* primary key field */
TA010013 TT010005 AAid student c8 1 /* fkey to student */
TA010014 TT010005 BBid course c8 1 /* fkey to course */

```

## 2.6 Output

CHGEN 10 generates the same \*.c and \*.h utility files that version 9.0 generates. The internal tables TT and TA are also output if the command line parameter “-metafile” is used. The resulting output file will be schemaname.msdat.

### 2.6.1 metaschema.dat

The *metaschema.dat* file contains the information from the schema file that populates the TT and TA tables. The top line of the file will be table SV(schema version) which identifies the source schema file and the type of schema file it is. The *metaschema.dat* file for the previous example might look like the following:

```

SV010001 PJ010001 schooldb.sch CHGEN CHGEN 9.0

TT010001 SV010001 AA students /* Table of student records */
TT010002 SV010001 BB roster /* Table associating students with courses */

TA010001 TT010001 AAid students_id c8 1 /* primary key field */
TA010002 TT010001 fname fname c30 0 /* first name of student */
TA010003 TT010001 lname lname c30 0 /* last name of student */

```

```
TA010004 TT010002 BBid roster_id c8 1 /* primary key field */
TA010005 TT010002 AAid student_id c8 1 /* foreign key field */
TA010006 TT010002 cname cname c30 0 /* name of course */
```

The SV table (first line) is organized in the following manner:

```
Schema number (always SV010001), Project number, schema filename,
Schema format (CHGEN or GENDB), version of CHGEN that created this file
```

The TT Table of Tables is organized in the following manner:

```
Table number, Schema number, two letter abbreviation, table name, comment
```

The TA Table of Attributes is organized in the following manner:

```
Attribute number, Table number, name, alt name, type, iskey?, comment
```

This example holds for both of the schema examples given with the exception of the first line (table SV). This is because it identifies the type and name of the schema file used as input. For these reasons, this line may be different when you attempt to run these examples through CHGEN v9.0. This specific example is for file schooldb.sch which is a GENDB type of schema file.

## 2.6.2 pr utilities

When CHGEN 10 runs successfully on either type of schema file, you will find at least six new files created in your directory. If you run CHGEN with the log option, seven new files will be created. If the schema file is named “students.sch”, the seven created files will be called:

```
students.h pr_delete.c pr_free.c pr_stats.c pr_dump.c pr_load.c pr_log.c
```

Note that the name of the schema file only changes the name of the “.h” file.

These files can now be used as modules of your application. The file “pr\_load.c” contains the routines; pr\_load(), pr\_add(), and pr\_count(). The file “pr\_log.c” contains the routines pr\_startlog(), pr\_stoplog(), and pr\_replay(), if present. The remaining “.c” files contain the same function as the name of the file (i.e. pr\_dump(), pr\_free(), etc.).

The routines defined within these files are the same as those expected from CHGEN v8.0. These routines are described in a later section “Using the pr\_ Routines” which is adopted from the CHGEN v8.0 user’s manual.

## 2.7 Limitations

There are some limitations to the types of schema CHGEN can support. Some of the known unsupported schema concepts are:

- Circular loops are not allowed (although support for this is planned for the next version of CHGEN). A circular loop occurs when a table lists itself as a parent. NOTE: The behavior of the code generated in this case is unpredictable, and this condition is NOT detected by CHGEN when using a CHGEN v8.0 style schema file. A circular loop in a GENDB style schema file will cause a fatal error.
- Union foreign keys. One of the concepts that is desired but not supported is the ability to define a foreign key like: AAid\_OR\_BBid\_OR\_CCid This would imply that this field points to either of three parent tables, AA, BB, or CC. NOTE: The code generated in this case will not compile. NOTE: This problem can be avoided by changing the field to be a datafield, rather than a foreign key (by changing the 1 to a 0 in the schema's is-key? field).
- Listing the same field (foreign key or data field) multiple times. NOTE: The code generated in this case will not compile, since multiply defined symbols would exist.

## 2.8 Error Conditions

CHGEN detects several warning and error conditions about your schema. In addition, CHGEN uses VMS-style error reporting. For those not familiar with this style of error message, consider the following:

```

CHGEN-F-FILNOTFND, can not open test.sch, file not found
^      ^      ^      ^      ^
|      |      |      |      |
|      |      |      +---- parameter.
|      |      |
|      |      +---- textual description of the error.
|      |
|      +----- symbolic name of the error condition.
|
|      +----- error severity (F=fatal, W=warning, I=info).
|
+----- application generating the error.

```

Warning and Informational messages generally indicate conditions that are not fatal, but should be corrected if possible. CHGEN continues to process your schema with these conditions. Fatal errors, however, indicate problems serious enough to prevent CHGEN from continuing. Schema syntax errors, duplicate table names, etc... are examples of such errors. The following lists all errors reported/detected by CHGEN:

Symbolic Name	Severity	Description
BADCMD	F	unknown value %s specified in command line.
BADDATE	F	data-type %s for field %s in table %s is not a valid date type.
BADFIELDNAME	F	invalid field name %s for table %s, must start with an alphabetic character.
BADFLOATTYPE	W	float data-type %s for field %s in table %s is invalid, value types are f4 and f8.
BADINTTYPE	W	integer data-type %s for field %s in table %s is invalid, value types are i2 and i4.
BADISKEY	F	'is-key' field %s for field %s in table %s must be 0, -1, or -s.
BADMAXVIEWS	F	%d is an invalid value for the maxviews qualifier.
BADQUAL	F	unknown qualifier %s specified in command line.
BADTBLNAME	F	invalid table name %s, must start with an alphabetic character.
BADTYPE	F	unsupported data-type %s specified for field %s in table %s.
CHARTEXT	F	char/text data-type %s for field %s in table %s has an invalid length specifier.
DUPTBLID	F	duplicate table-id (%s) found in schema.
DUPTBLID	F	duplicate field-name (%s) found in table %s (%s).
DUPTBLNAME	F	duplicate table-name (%s) found in schema.
ERROPENMSFILE	F	The <i>metaschema.dat</i> file could not be opened.
ERRNOPARENT	F	Parent table not found (May be a forward reference).
ERROPNOUT	F	can not create output file %s.
EXTRABEGIN	F	extra '{' found for table %s.
EXTRAEND	F	found an extra '}' when expecting the start of a new table, last table processed was %s.
FILENAMELEN	F	input schema filename length (%d) exceeds the max of %d.
FILNOTFND	F	can not open %s, file not found.
FIRSTMUSTBEKEY	F	first field %s for table %s must have a 'is-key' value of 1.
FORWARDREF	I	forward reference to table %s (%s) found in table %s (%s) via foreign key field %s.
INVARGS	F	Usage: chgen ;schema-filename;.
KEYMUSTBEC8	F	primary key field %s for table %s must be of type c8.
KEYMUSTBEC8	F	foreign key field %s for table %s must be of type c8.

MISSINGBEGIN	F	missing '{' for table %s.
MISSINGCOMMENT	W	table %s does not have a comment.
MISSINGCOMMENT	W	field %s for table %s does not have a comment.
MISSINGTBLLINE	F	found '{' when expecting the start of a new table, last table processed was %s.
MISSINGTBLSTART	F	The keywords "create table" not found where expected.
NAMETOOLONG	F	table name %s exceeds the maximum length of %d.
NAMETOOLONG	F	table abbreviation (%s) for table %s must be 2 characters long.
NAMETOOLONG	F	field name %s for table %s exceeds the maximum length of %d.
NAMETOOLONG	F	alternate field name %s for field %s in table %s exceeds the maximum length of %d.
NOBPFOREIGNONLY	F	the no-backpointer flag only makes sense for foreign keys (field %s for table %s)
NONDEFAULTQUAL	I	Using non-default command-line qualifier -ansi
NONDEFAULTQUAL	I	Using non-default command-line qualifier -nobp
NONDEFAULTQUAL	I	Using non-default command-line qualifier -noforward
NONDEFAULTQUAL	I	Using non-default command-line qualifier -maxviews
NOPATHS	W	paths are supported by GENDB but not this version of CHGEN.
NOSCHEMAFILE	F	No schema file specified
PKEYABBREVNOMATCH	F	first 2 characters of primary key field %s for table %s must equal the table abbreviation %s.
TYPETOOLONG	F	data-type %s for field %s in table %s exceeds the maximum length of %d.
UNKNOWNBLEREF	F	unknown table %s referenced as a foreign key (%s) in table %s (%s).

## 2.9 Command Line Switches

CHGEN supports command line switches, similar to many UNIX applications. The complete CHGEN calling format is :

```
chgen schemafile [ -nobp ] [ -ansi ] [ -validate ]
```

NOTE: If a file extension is not specified for the schemafile, .sch is assumed.

**-metafile** This qualifier causes CHGEN to output The values of tables TT and TA to an output file. The output file is called schemaname.msdat.

**-gendbschema** This qualifier causes CHGEN to expect a schema file in the GENDB format. Using

this qualifier with a CHGEN v8.0 schema file format will generate parsing errors.

- keysize=N** This qualifier specifies the external character length of all keys. Valid values for N are 8 and 12. The resulting .h file will contain two defined symbols, HCG\_KEY\_SIZE, which is equal to N, and HCG\_ABBR\_SIZE, which will be 2 or 4 depending on the value of N (8 and 12, respectively). All table abbreviations present in the schema file and view definition file must be of size HCG\_ABBR\_SIZE, and all keys present in any data files called by pr\_init or pr\_load must have a character length of HCG\_KEY\_SIZE. If no keysize qualifier is specified to CHGEN, a default of 8 characters is used.
- ansi** This qualifier will cause chgen to use ANSI "C" identifier concatenation symbols in the resulting .h file, rather than the non ANSI symbol. In ANSI "C", two identifiers are concatenated using the ## symbol. In non-ANSI "C", the symbol is /\*\*/. You may want to use this qualifier for the GCC and GPP compilers (previously, you had to use the -traditional qualifier for those compilers to get the /\*\*/ symbol to work). The default for this concept is non ANSI.
- nobp** This qualifier will cause chgen to output NO backpointers, regardless of what is specified in the is-key field of the schema file. The default (ie, without this flag) is to generate backpointers except where specified in the schema file.
- quiet** This qualifier will cause chgen to NOT report informational and warning messages when compiling the schema. See section I.E for error classifications.
- validate** This qualifier will cause chgen to only validate the schema file. No output files will be generated.
- noforward** This qualifier will cause chgen to write code that assumes there are no forward references in the data being loaded. A forward reference occurs when a child row is loaded before the parent is loaded. Using this qualifier will reduce the code size requirements and speed up applications (no link\_child calls are made inside pr\_add). To help avoid integrity problems, a special check is made when a child is loaded to ensure that the parent is already loaded. This check is only made when the -noforward qualifier is used.
- noorder** This qualifier will cause chgen to NOT bother storing table rows in primary-key sorted order. This will probably not be a problem, since applications rarely care about the actual values of the primary keys. In addition, children rows will NOT be stored in data-load order in the child chains (ie, along the fpp chain). In fact, the children will be linked in a reverse data-load order. Using this qualifier will lead to further code size reductions and speed improvements, but should not be used if data order is important to your application.
- maxviews=n** This qualifier will cause chgen to generate code that will allow up to the specified number of views to be defined. If this qualifier is not used, the default maximum of 10 views will be used.
- log** This qualifier will cause chgen to generate code that implement the logging routines "pr\_replay()", "pr\_startlog()" and "pr\_stoplog()" in the file pr\_log.c. Logging support will not be enabled otherwise.

## 3 Writing Applications

### 3.1 Include Files

Each module of your application that needs to use the network data- structures MUST include the application header file (eg. schoolb.h). The following program stub demonstrates how to do this:

```
#include <stdio.h>
#include "schoolb.h"
```

```
main()
{
}
```

<<<<< IMPORTANT >>>>>

In addition, you MUST place the following line as the very first line of your main program :

```
#define MAIN
```

Failure to do this will cause the "pr" routines to fail at run-time.

### 3.2 "C" Files

Your application's make/build scripts should compile the five (or six if the log qualifier is used) generated "C" files (pr\_load, pr\_dump, pr\_free, pr\_delete and pr\_stats (and pr\_log if logging is enabled)) as if they were modules of your application (in fact, they are!). Similarly, you must link with them, just as you would with any other modules. As stated above, you only have to link with the modules you actually use.

### 3.3 Global Variables

#### i. Reserved Words -

Your application should avoid declaring any symbols that start with the letters "hcg-", since this is the reserved prefix used internally by the routines.

#### ii. Table Variables -

There is a set of variables declared for each table in the schema. A hypothetical table AA is referred to in the following:

**AA** This variable is a pointer to the first row of the AA table. Your application must NEVER write to this variable, but can reference it freely.

**AAcurr, AAtemp** Temporary pointer variables used by the various pr\_ routines. These can be used by your application freely, and will not interfere with the internal workings of the routines.

**AAend** This variable points to the last row of the AA table. Like the AA variable, you must NEVER write to this variable, but it can be referenced.

**AA\_elt** This variable is a pointer to the most recently added row to the AA table. Again, do not write to this variable.

**AA\_idx** This variable is not a pointer, but is an integer index into an internal list of tables. Although applications do not need to reference this, it is mentioned here since it must be passed along to some of the routines (discussed later).

## 4 Datafile Formats

### 4.1 General Format

Data representing a database is stored in datafiles. A datafile is composed of individual lines, and each line describes one row of a database table. For example, consider the following simple schema for a database with tables AA (students) and BB (courses taken by students):

```
students AA
{
  AAid student_id c8 1
  fname first_name c30 0
  lname last_name c30 0
}

roster BB
{
  BBid roster_id c8 1
  AAid student_id c8 1
  cname course_name c30 0
}
```

A datafile for this schema might be :

```
AA030001 John Doe
AA030002 Sue Smith
BB030001 AA030001 Calculus
BB030002 AA030001 Physics
BB030003 AA030002 Robotics
BB020001 AA030001 Chemistry
```

**NOTE:** The restriction that each line MUST begin with a single space has been removed.

**NOTE:** A data line is considered to be a set of fields that map one- to-one to fields in the appropriate database table. Each field is seperated by at least one space. Missing or extra fields in a data line can cause incorrect values to be loaded. For example, if a string appears

where a float is expected, the "C" atof function will return zero (0), and a 0 will be stored in the network data-structures.

**NOTE:** For those readers fluent in the ways of relational databases, you may have noticed that the example used here is not in 3rd normal form! Technically, there should be a CC table listing the courses that exist, and the BB table should simply have a foreign key (CCid) that points to the course a student is taking.

## 4.2 String Fields

A string field (of a database table) CANNOT contain spaces because spaces are used to separate fields in a data line. In the example above, 'John' and 'Doe' are considered to be two separate string fields. Fortunately, the schema for the example has taken this into account by having separate fname and lname fields. As of release 4.0 of CHGEN, the last field of a database table CAN have embedded whitespace, if it is a string field. In the example above, the schema could have had one field called sname, which contains both the first and last names. Since it would be the last field in the record and it is a string, it would work.

# 5 Using the pr\_ Routines

## 5.1 pr\_init

The pr\_init routine is used to initialize the network data-structures, define all views that might be used, and identify ALL datafiles that exist for the database. This MUST be done before calling any other "pr" routines. The calling format is :

```
pr_init( view-definition-file, list-of-data-files );
```

where

**view-definition-file** This string parameter specifies a file which defines all views that might be used in the application. This file will be opened, parsed, and closed. See section 9 for a complete description of this file and its format.

**list-of-data-files** A string representing the list of ALL data files that exist for this database, even if the application will not be using them. The list itself is composed of filenames (which may include directory specifications), separated by spaces. Each file will be opened, scanned quickly to gather table statistics, and closed. This process is critical to the integrity of the data itself. In the future, this phase of gathering table statistics may be changed (to read directly from an Ingres database for example).

The following is an example of a complete call to pr\_init:

```
pr_init("students.viewdefs", "students90.dat students89.dat students88.dat");
```

**NOTE:** Your application must call pr\_init before calling any other "pr" routines.

**NOTE:** Your application can call `pr_init` more than once provided it calls `pr_free` first (see below). This should not be necessary however, since all views are already known, and table statistics are still "fresh" in internal memory (unless `pr_free` has in fact been called).

**NOTE:** If `pr_init` encounters a version that is greater than the maximum allowed (99 max for 8 character keys, 255 max for 12 character keys), then CHGEN will exit because this data is not valid and can not be used. If `pr_init` encounters a version that is equal to the maximum allowed, and a mode of 'write' has been requested for this table, then CHGEN will exit since this data is not updateable.

## 5.2 `pr_load`

The `pr_load` routine is used to populate the network data-structures from a data file. `pr_init` must be called before calling this routine. The calling format is :

```
pr_load( viewname, filename );
```

where

**viewname** A string representing the name of the view to use when loading the data. Any data not meeting the view will be ignored. Data will not meet the view if it is either a table not in the view, or the table is ok but the version number is wrong.

**filename** A string representing the name of the datafile to be loaded. The file specified must follow the format described in section III.

The following is an example of a complete call to `pr_load`:

```
pr_load("RosterView","students90.dat");
```

**NOTE:** `pr_load` automatically links all parent and child pointers (i.e., assigns proper pointer values to all of the fields in the network data-structures).

**NOTE:** The data can be loaded in any order, and does not have to be sorted (`pr_load` adds the data into the internal tables in sorted order). Unless the `-noforward` qualifier was used when CHGEN was run, it is completely acceptable to load child rows that reference parents that are not yet loaded. When the parent row is finally loaded, it will link itself in correctly. If the `-noforward` qualifier was used, however, parents must be loaded before children.

**NOTE:** If a child row is loaded, and it specifies a parent that is not loaded, no errors are generated. In this case, the parent pointer field in the child row is NULL, and referencing it will cause a memory access violation! It is up to the application to handle data integrity issues such as this. If the `-noforward` qualifier was used, a warning will be issued when the child without a parent is loaded.

**NOTE:** Your application can call `pr_load` multiple times. This feature has been created so that applications can store each tables data in its own file, if necessary.

**NOTE:** `pr_load` will print a warning message if a duplicate row is encountered. The duplicate row will NOT be added to the network data-structures.

**NOTE:** All data for a view does not have to be in the same file (it can be spread out). Similarly, all data for all versions, tables, etc... could all be stored in one file.

**NOTE:** If a table set as binary tree search, it will build b-tree in `pr_load`.

### 5.3 `pr_create`

The `pr_create` routine is used to allocate and initialize memory for the table element pointer. It returns a pointer to the table element created. The calling format is :

```
pr_create( tbl_abbrev );  
where
```

**tbl\_abbrev** The table abbreviation defined in the user schema file.

**NOTE:** Applications should no longer use this function and should use `pr_create` instead. The `pr_gen_pkey` routine is used to generate a value for the next available primary key for the specified table. The function is still used by CHGEN itself in `gen_pr_load.c`. Internally, this routine locates the primary key of the last row in the table, increments the last 4 digits, and returns this value. If the table is empty, the returned value would be "AA030000" (for example). Again using the table AA as an example, the calling format is :

```
pr_gen_pkey("RosterView",AA,temp);  
where
```

**"RosterView"** The view to apply. The version number of this table specified by this view will be used.

**AA** Specifies the table for which a new primary key value is requested.

**temp** This is the returned string (c8) that holds the next available primary key value. This would probably be a temporary variable of the application.

The following is an example of a complete call to `pr_create`:

```
struct AA *AA_elt;  
AA_elt = pr_create(AA);
```

### 5.4 `pr_add`

The `pr_add` routine is used to add (append) a new row to one of the tables in the network data structure. `pr_init` must be called before calling this routine. The calling format is :

```
pr_add( viewname, tbl_abbrev, tbl_elt );  
where
```

**viewname** A string representing the name of the view to use when loading the data.

**tbl\_abbrev** The table abbreviation defined in the user schema file.

**tbl\_elt** The table element pointer.

The following is an example of a complete call to `pr_add`:

```
struct AA *AA_elt;
pr_add("RosterView", MA, MA_elt);
```

**NOTE:** `pr_add`, like `pr_load`, will ignore the data if it does not meet the view.

**NOTE:** Calling `pr_add()` with a table that is not in the view will NOT add the table to the view (which was the case in previous versions of CHGEN).

**NOTE:** `pr_add` will print a warning message if a duplicate row is encountered. The duplicate row will NOT be added to the network data-structures.

**NOTE:** `pr_add` will print a warning message if the user did not assign values for the foreign keys or did not call `pr_set_default(1)`.

**NOTE:** `pr_add` will print a warning message if the table element being added contains an unassigned non-key field of type `cX` or `tX`.

## 5.5 `pr_dump`

The `pr_dump` routine is used to output a new copy of all loaded data for a given view. The output file follows the standard datafile format. `pr_init` must be called before calling this routine. This routine is also the only way to increment the version number of the data. The calling format is :

```
pr_dump( viewname, filename, new-version-flag, file-mode);
```

where

**viewname** A string representing the name of the view to use when outputting the data. The mode of the view (read, write, update) along with the `new-version-flag` dictates that a new version of the output data is desired.

**filename** A string representing the name of the output datafile to be generated. All data from the network data-structures meeting the view will be printed to this file. If the file already exists, it will be overwritten (or a new version created, if the host system is VMS) if the `file-mode` is "w").

**new-version-flag** This integer flag (0=false, 1=true) indicates whether a new version number should be output for each data row. If this flag is 1, AND the view has a mode of "write", then each row will have its version number changed to one higher than the highest version number that exists for that table.

**file-mode** This string parameter indicates whether a new output datafile should be created, or an append to an existing datafile should be done. This mode value is the same as the one you

pass to the "C" routine fopen, ie:  
"w" : Create a new output datafile.  
"a" : Append to an existing datafile.

The following is an example of a complete call to pr\_dump:

```
pr_dump("RosterView","mydatabase.dat",1,"w");
```

Assuming the datafile that was loaded by pr\_load is the one specified in the example above, and the sample pr\_add was also done, then a new output file will be created (overwriting the old under UNIX, creating a new version of the file under VMS) :

```
AA040001 John Doe  
AA040002 Sue Smith  
AA040003 Davie Jones  
BB040001 AA040001 Calculus  
BB040002 AA040001 Physics  
BB040003 AA040002 Robotics  
BB040004 AA040002 Physics
```

Note that the row that was ignored (since it was for version 2) is not listed, add the newly added row is present. Also note that the version number for all keys is now 4.

**NOTE:** pr\_dump will NOT update the version number of any foreign keys that point to tables that are not in the view. This concept is vital to the integrity of version numbers. For example, if version 3 of tables AA, BB, and CC are loaded, and table CC has a parent DD that is not loaded, then the version of the DD table that was referred to at load time is still the same version (since a new version of DD was not generated).

**NOTE:** The order of the rows appearing in the output file may not be the same as the order they appear in the input file for two reasons. First, pr\_load (and pr\_add) sort the rows (unless the -noorder qualifier was used when CHGEN was run), so they will be output in sorted order. If the input file was not sorted, there will be a difference in the appearance of the files. Second, the tables are output in the order they are defined in the schema (.sch) file, which may not be the same order that they appear in the input data file. Since pr\_load is immune to the order of data in an input file, this should not be a concern (the files are still logically equivalent).

**NOTE:** The mode string is really just the mode string taken by the "C" routine fopen(), and pr\_dump passes the string directly to fopen. Feel free to get as fancy as you want with this value, although the code may fail if you use a bad mode string.

## 5.6 pr\_dump\_row

The pr\_dump\_row routine is used to write-out (dump) a single row that meet the specified view to the specified file-name. The calling format is

```
pr_dump_row(tbl, viewname, filename, new-version-flag, file-mode);
```

where

**tbl** The table defines in the user schema file.

**viewname** A string representing the name of the view to use when outputting the data. The mode of the view (read, write, update) along with the new-version-flag dictates that a new version of the output data is desired.

**filename** A string representing the name of the output datafile to be generated. All data from the network data-structures meeting the view will be printed to this file. If the file already exists, it will be overwritten (or a new version created, if the host system is VMS) if the file-mode is "w").

**new-version-flag** This integer flag (0=false, 1=true) indicates whether a new version number should be output for each data row. If this flag is 1, AND the view has a mode of "write", then each row will have its version number changed to one higher than the highest version number that exists for that table.

**file-mode** This string parameter indicates whether a new output datafile should be created, or an append to an existing datafile should be done. This mode value is the same as the one you pass to the "C" routine fopen, ie:

"w" : Create a new output datafile.  
"a" : Append to an existing datafile.

The following is an example of a complete call to pr\_dump\_row:

```
if(encode("AA030003", &i)==1)
{
pr_find(AA, AAid, i);
if(AAcurr==NULL)
printf("Student 'AA030003' not found, try again.\n");
else
pr_dump_row(AA, "RosterView", "test.dat", 1, "w");
}

find_str_loop("RosterView", AA, fname, "Jon")
pr_dump_row(AA, "RosterView", "test.dat", 1, "w");
```

## 5.7 pr\_free

The pr\_free routine is used to release the memory held by the network data structures, and to blank them out. This MUST be called if you wish to start another session of processing (ie, before calling pr\_init again). The calling format is :

```
pr_free();
```

**NOTE:** Applications do not have to call `pr_free()` if they only call `pr_init` once.

## 5.8 `pr_delete`

### i. General Usage -

The `pr_delete` routine is used to delete (and release the memory held by) a single data row in a table. Given a table name, it deletes the row pointed to by the `XXcurr` variable. The `pr_delete` function also 'unlinks' the row from all parents and children linked to it. Doing this will 'strand' all children (although their pointer variables referring to the deleted parent are safely blanked out). Using the table `AA` as an example, the calling format is :

```
pr_delete(AA);
```

The following program fragment shows a typical call to `pr_delete`:

```
find_str_loop("RosterView",AA,lname,"Jones")
if (strcmp(AAcurr->fname,"Davie") == 0)
pr_delete(AA);
```

### ii. Recursive Deletes -

It is possible (although not necessarily easy) to delete all children of a row (and all its children...) when the row is deleted. No macros or routines are provided for this (although they may in the future), so the following template shows how it could be done manually. Suppose the schema is a simple  $AA \rightarrow BB \rightarrow CC$  relationship, and the user wants to delete an `AA` row (and all its descendants). The following code shows how this would be done:

```

/*****
/* Assuming AAcurr points to the AA row to be deleted, loop over the BB */
/* children.          */
*****/
child_loop(AA,BB,BBid,AAid)
{
/*****
/* Now, perform a 'delete loop' over the CC children of this BB row. */
*****/
child_loop(BB,CC,CCid,BBid)
{
/*****
/* Next, save a pointer to the 'next' CC child under the BB row.    */
*****/
CCTemp2 = CCcurr->BBid_fpp;
/*****
/* Actually delete the CC row.          */
*****/
pr_delete(CC);
```

```

/*****
/* Now, restore the CCcurr variable from the saved pointer.      */
/*****
CCcurr = CCtemp2;
}
/*****
* This note was added 97/10/10 by RJL at line 1521 in Section 5.8(ii)
* (Recursive Deletes) of
*      $CASE/gen/ver_10log/doc/gen10_manual.tex:

* Whether the above solution works depends on how pr_delete updates CCcurr.
* (Is it set to the predecessor of the deleted CCcurr or to its successor,
* or does it now point to possible garbage in the deallocated space?

* If CCcurr becomes the predecessor of the deleted element, then
* nothing needs to be done; pr_unlink updates the predecessor's BBid_fpp.

* If CCcurr becomes the successor of the deleted CCcurr, then the
* child_loop will step over this value by doing CCcurr = CCcurr->BBid_fcp.
* Then pr_delete in the child_loop will only delete every other CC-child.

* Now consider the case where CCcurr is undefined after the pr_delete:
* Replacing CCcurr by its successor will not work if the child_loop will
* step CCcurr to CCcurr->BBid_fpp a second time when it iterates,
* thereby skipping over one row.

* A possible strategy (not tested) is 'one step back, 2 steps forward':
* BEFORE the child_loop insert "CCprior = BBcurr;"
* (Save the predecessor of CCcurr, initially the child_loop parent)
*
* AFTER the pr_delete call inside the loop, insert:
*      "CCcurr = CCprior;
*      CCprior = CCcurr->BBid_fpp;"
* (Restore CCcurr to CCprior, then update CCprior so it again points to
* the successor of CCprior. The rationale behind this is that pr_unlink
* will update the value of CCprior->BBid_fpp to the CC-row beyond the
* deleted one, to bridge the gap left by the deletion of CCcurr.
* If CCcurr = CCprior is followed by the stepping action of
* the next child_loop iteration, the latter will make the next value
* of CCcurr = the value of the now-deleted oldCCcurr->BBid_fpp.

* There is a special case here when deleting the FIRST child:
* As the child_loop recycles it does:
*      newCCcurr = CCcurr->BBid_fpp (= oldCCprior->BBid_fpp);
* and oldCCprior was initialized to BBcurr;
* If pr_unlink unlinked the first child properly, it will have updated
* BBcurr->CCid_fcp to the second CC-child of BBcurr, as desired.
*

```

```

* - end Note RJL 97/10/9
*/

/* Another comment: the unlink macros seem to make a ring structure
* of the child-lists as long as the list is not empty. However if
* the parent has NO children, the parent's CCid_fcp (and CCid_bcp)
* is set to NULL instead of pointing to itself. This is not as
* consistent as a self-loop (empty ring). Which requires more special
* treatment? - RJL 97/10/10
*/

/*****
/* Now that all the CC children of this BB row are gone, delete the BB*/
/* row after saving the BBcurr pointer (just like above).      */
/*****
BBtemp2 = BBcurr->AAid_fpp;
pr_delete(BB);
BBcurr = BBtemp2;
    }
    /*****
    /* Finally, delete the AA row!      */
    /*****
    pr_delete(AA);

```

**NOTE:** If a table set as btree search, it will delete a row in btree.

## 5.9 pr\_find

The pr\_find routine is used to locate a particular row in a table. Using the table AA as an example, the calling format is :

```
pr_find(AA,AAid,pkey);
```

where

AA : Specifies which table is to be searched.

AAid : Tells pr\_find the name of the pkey for the table to be searched.

pkey : The primary key value (must be encoded first) to be searched for.

The AAcurr variable holds the result of the search. If the specified row is NOT found, then the variable will be NULL. If the find was successful, then the variable points to the specified row.

The following program fragment shows a typical call to pr\_find:

```

if(encode("AA030003", &i)==1)
{
pr_find(AA, AAid, i);
if(AAcurr==NULL)

```

```
printf("Student 'AA030003' not found, try again.\n");
else
printf("The name of student 'AA030002' is %s %s\n,
AAcurr->fname, AAcurr->lname);
}
```

## 5.10 pr\_rcount

The `pr_rcount` routine returns the number of rows loaded for the specified table, for the specified view. Again using the table `AA` as an example, the calling format is :

```
x = pr_rcount("RosterView",AA);
```

where

"RosterView" : The view to apply. Any rows in this table with a different version number than specified by this view will be ignored. `AA` : The name of the table whose row count is requested.

The following program fragment shows a typical call to `pr_rcount`:

```
printf("The number of rows in the 'AA' table is "RosterView",pr_rcount(AA));
```

**NOTE:** To count the number of loaded rows in the entire table (ie, regardless of version number), use the `pr_rcount_all()` macro. This macro does not take in a viewname.

## 5.11 pr\_stats

The `pr_stats` routine is used to output a listing of some basic information about the schema, defined views, and loaded data, to the specified data file. Data includes view listings, table rowcounts and `maxmkeys` (for any loaded version), and allocation sizes for each table table. Also included is the current size of the overhead data-structures used internally by the system. The calling format is :

```
pr_stats( filename, file-mode);
```

where

**filename** A string representing the name of the statistics file to be generated.

**file-mode** This string parameter indicates whether a new statistics file should be created, or an append to an existing file should be done. This mode value is the same as the one you pass to the `pr_dump`, ie:

- "w" : Create a new output file.
- "a" : Append to an existing file.

The following is an example of a complete call to `pr_stats`:

```
pr_stats("students.stats","w");
```

## 5.12 pr\_startlog

The `pr_startlog` routine starts a logging session, where database operations are logged to a file for later replay. This routine is available only if `chgen` was run with the `log` flag. The routine calls `pr_dump` to take a snapshot of the current state of the database for use in the later replay, saving it in a file named `fileDB1.dat`, where `file` is the file name passed in the `file_name` parameter described below. `pr_startlog` returns an `int`.

The calling format is `pr_startlog(file_name, viewname)`

where

**file\_name** A char \* representing the base name of the file where the log is to be placed. The log file name is formed by adding to the `file_name` parameter the suffix `.txt`. For the format of the log file, see the Genlog Final Report.

**viewname** A char \* representing the view to use to make the log. Note that the view must be “wide” enough to include every table used in the session.

If it completes successfully, `pr_startlog` returns 0. If it does not, `pr_startlog` will return

- 1 `pr_startlog` was called when the database had not yet been initialized (before `pr_init` was called).
- 2 `pr_startlog` was called with an invalid view (one that was not defined in the `viewdefs` file used in `pr_init`).
- 3 `pr_startlog` was called while logging was on, that is `pr_startlog` was called after an earlier `pr_startlog` was called but before `pr_stoplog`.
- 4 `pr_startlog` was called with a null `file_name` parameter.

`pr_startlog` will print the following error messages when it encounters an error:

**Error: pr\_startlog called when database is not yet initialized.** `pr_startlog` had been called before `pr_init`.

**Error: view passed to pr\_startlog is not defined.** The view indicated is not in the `viewdefs` file.

**Error: pr\_startlog called when logging was not off.** `pr_startlog` had already been called.

**Error: log file name null; logging not on.** The logfile name was not set.

**Error: pr\_startlog() cannot open %s** The routine could not open the logfile.

## 5.13 pr\_stoplog

This routine finishes a log session. It causes an end of log record to be written to the log file, and releases all resources used internally in making the log file. If instructed to do so, `pr_stoplog` will call `pr_dump` to take a snapshot of the database at the time logging stopped. `pr_stoplog` returns an int.

The calling format is `pr_stoplog(view_name, dump_flag)`

where

**view\_name** A char \* variable representing the view name to use if taking a snapshot `pr_dump`.

**dump\_flag** An int variable indicating whether to take a database snapshot. A value of non-zero means that a database snapshot should be taken, zero that it should not be taken. `pr_stoplog` will invoke `pr_dump` with a filename parameter of `fileDB2.dat` (where file is the `file_name` parameter of the latest `pr_startlog` call) if a snapshot is to be taken.

If it ends successfully, `pr_stoplog` returns 0. If it encounters an error, `pr_stoplog` returns

- 1 `pr_stoplog` was called when logging was not active. That is, `pr_startlog` had not been called.
- 2 The `view_name` parameter referred to an invalid view.

`pr_stoplog` will print the following error messages when it finds a fault condition:

**Error: `pr_stoplog()` called when logging not active.** `pr_startlog` had never been called.

**Error: view %s passed to `pr_startlog` is not defined.** The named view is not in the `viewdefs` file.

## 5.14 pr\_replay

The `pr_replay` routine replays the log session generated by an earlier set of `pr_startlog` `pr_stoplog` invocations. The general technique used is to

1. open the logfile.
2. clear the current database with `pr_free`.
3. use the database snapshot generated by `pr_startlog` to initialize the database.
4. for each entry in the log file, file replay the command. If so indicated, `pr_replay` will simulate the time interval between the original database operations by waiting. See the “`near_realtime`” parameter below.

The function has the calling sequence `pr_replay(view, log_file, near_realtime, take_snapshot)`. The parameters are described below.

**view** A char \* representing the view to use to when initializing the database. The view must be “broad enough” to include all tables used in the session.

**log\_file** A char \* representing the name of a log session file to replay. This file must have been produced by an earlier `pr_startlog` `pr_stoplog` invocation. See the *CHGEN Log Final Report* for a description of the log file format.

**near\_realtime** An integer indicating that the replay should approximate the time interval that occurred between database operations in the original session. 0 indicates that the log should not be played back in near realtime, but should be played back as quickly as possible. 1 indicates that the log should be played back in near realtime. A value other than 0 or 1 will be treated as 1, but will cause an error message.

**take\_endsnapshot** An integer indicating whether to take a snapshot of the database after the replay with `pr_dump`. A value of 1 means to take the snapshot. A value not equal to 1 means not to. If indicated, the snapshot is placed in a file name `fileDB3.dat`, where `file` is the value of the `logfile` parameter.

`pr_replay` returns an integer. 0 indicates success. Other values indicate a failure condition. They are

- 1 The view name was null.
- 2 The log file name was null.
- 3 `pr_replay` could not open the log file.
- 4 A log session was already in progress; that is, `pr_stoplog` had not yet been called.
- 5 `pr_replay` could not interpret the log file. The log file specified in the parameter list has either been corrupted, or was not a log file to begin with.

`pr_replay` will print the following error messages when it finds a fault condition:

**pr\_replay error: logging in progress** `pr_replay` found that it was already in the process of replaying a log.

**pr\_replay error: null view name** The view name parameter is empty.

**pr\_replay error: null log file** The log file name parameter is empty.

## 6 Key Manipulation Functions

### 6.1 decode

The `decode` function has been provided for the purpose of decoding an integer key into its character equivalent. This is necessary when the database is stored via `pr_dump`, and also may be required when the application programmer is adding foreign keys to a new table row. The calling format is:

```
hcg_key key_int;  
decode(key_char, &key_int);
```

**key\_char** This parameter should be a pointer to a character string of minimum size HCG\_KEY\_SIZE+1. Key\_char will contain the character equivalent of key\_int if the decode is successful.

**key\_int** This parameter should be a pointer to an integer containing the encoded version of a character key.

**NOTE:** Decode requires a pointer to both arguments. Since the keys contained in memory at runtime are not declared as pointers, this requires passing the address of the key to decode.

Decode will return a status of 1 is successful or -1 if the integer key is invalid. The following is an example of a complete call to decode:

```
if(decode(temp_key, &AAcurr->AAid) != 1)  
printf("Error: AAcurr->AAid contains invalid key!\n");  
else  
printf("AAcurr->AAid is equal to %s\n", temp_key);
```

## 6.2 decode\_retstr

The decode\_retstr function has been provided as a convenience function to be used when a key's value is only required once. This function returns a static character string containing the key's value if successful. This is useful because a temporary key value does not have to be declared and the call can be embedded within functions such as printf and sprintf. The calling format is:

```
decode_retstr(&key_int);
```

**key\_int** This parameter should be a pointer to an integer containing an encoded key value. Since all keys are not stored internally as pointers, the & operator is required.

**NOTE:** Because this function returns a static string, the next call made to this function will wipe out the previous value returned by it. This function CAN NOT be used more than once inside functions with unpredictable parameter assigning, such as printf and sprintf. If more than one key value needs to be decoded, use the decode function listed above with temporary key values.

The following is an example of a complete call to decode\_retstr:

```
printf("The primary keys contained in table AA are:\n");  
table_loop("RosterView",AA)  
{  
    printf("%s\n",decode_retstr(&AAcurr->AAid));  
}
```

### 6.3 encode

The encode function can be used to encode a character key to obtain its integer value. This function is used by pr\_add when adding a new row to a table. This function may be used by the application programmer if the programmer feels it is necessary. The calling format is:

```
encode(key_char, &key_int);
```

**key\_char** This parameter is a pointer to a character string of minimum size HCG\_KEY\_SIZE+1. Key\_char is the source for the encode function and must contain a key consisting of HCG\_KEY\_SIZE characters.

**key\_int** This parameter is a pointer to a key of data type 'hcg\_key'. This type is defined in the .h file and may be used by the application programmer if required. Assuming that key\_int is not declared as a pointer, its address must be passed to encode.

Encode returns a status of 1 if successful and -1 if key\_char contains an invalid character key. The following is an example of a complete call to encode:

```
if(encode(key_char, &key_int) != 1)
printf("Error: key_char is not a valid key: %s\n",key_char);
```

### 6.4 key\_compare

The key\_compare function can be used to compare two encoded key values. This function is used in several places by the CHGEN generated code to perform comparisons. Assuming that the application programmer were to use the encode function above, it is likely that they would also want to use this function. The calling format is:

```
key_compare(&key1, &key2)
```

**key1, key2** The parameters are two pointers to two encoded key values to be compared. Assuming the keys are not declared as pointers, the & operator must be used to pass their addresses to key\_compare.

Key\_compare returns a status identical to that of strcmp: A value less than zero if key1 < key2, 0 if key1 = key2 , and a value more than zero if key1 > key2. The following is an example of a complete call to key\_compare:

```
/** Check table AA to see if a primary key is already present */
if(encode(new_key, &new_int_key) == 1)
{
table_loop("RosterView",AA)
```

```

{
    if(key_compare(&new_int_key, &AAcurr->AAid) == 0)
        flag = true;
}
if(flag)
    /** Add a row with new_int_key or something along those lines **/
}

```

## 7 Iterators and Other Macros

### 7.1 table\_loop

The `table_loop` macro (and all the other "loop" macros) have been provided to simplify the code required to traverse tables within the network data structures. Using these macros, an application does not have to deal with the various internal pointer names to navigate between parents and children. The calling format is:

```

table_loop("RosterView",AA)
{
    <your code here>
}

```

where

**"RosterView"** This parameter indicates which view to apply (to determine which version number of the tables data rows).

**AA** This parameter represents the name of the table to be traversed. The user specified block of code will be run once for each row in the specified table meeting the view. There is nothing magical about this macro, it really maps to a for loop over the table. With each row, use `AAcurr` (ie, the "curr" table variable for the specified table) to reference the current row. As with any "C" program block, you do not need the curly brace pair if only one line is to be executed for each row.

The following is an example of using the `table_loop` macro:

```

printf("the list of students is:\n");
table_loop("RosterView",AA)
    printf("    %s %s\n",AAcurr->fname,AAcurr->lname);

```

**NOTE:** To traverse an entire table (ie, regardless of version number ), use the `table_loop_all()` macro. This macro does not take in a viewname.

## 7.2 child\_loop

The `child_loop` macro works the same as `table_loop`, but it is used to traverse all children of a given parent. You must specify the name of the parent and child tables, and the names of the foreign key fields needed to locate the appropriate child. The calling format is:

```
child_loop(AA,BB,BBid,AAid)
{
  <your code here>
}
```

where

**AA** This parameter represents the name of the parent table. The actual parent row being used is the one currently pointed to by `AAcurr`.

**BB** The name of the child table under the parent. All `BB` rows that have `AA` as their parent will be traversed.

**BBid** The name of the foreign key field that points downwards from the parent to the child. This tells the macro which pointer chain to follow to reach the `BB` rows.

**AAid** The name of the foreign key that points upwards from the child to the parent. This tells the macro which parent pointer chain to follow through the `BB` rows.

The following is an example of using the `child_loop` macro:

```
printf("%s %s's courses are:\n",AAcurr->fname,AAcurr->lname);
child_loop(AA,BB,BBid,AAid)
  printf("  %s\n",BBcurr->cname);
```

## 7.3 first\_child

The `first_child` macro is a handy way to tell if you are looking at the first child under a parent row. One possible use for this is to print a table header if you are about to print the first child. The calling format is:

```
if (first_child(AA,BB,BBid))
{
  <your code here>
}
```

where

**AA** This parameter represents the name of the parent table. The actual parent row being used is the one currently pointed to by `AAcurr`.

**BB** The name of the child table under the parent.

**BBid** The name of the foreign key field that points downwards from the parent to the child. This tells the macro which pointer chain to check to see if the current BB row is the first BB child under the AA row.

The following is an example of using the `first_child` macro:

```
printf("%s %s's courses are:\n",AAcurr->fname,AAcurr->lname);
child_loop(AA,BB,BBid,AAid)
{
  if (first_child(AA,BB,BBid))
  {
    printf("Course\n");
    printf("-----\n");
  }
  printf("  %s\n",BBcurr->cname);
}
```

## 7.4 last\_child

The `last_child` macro is a handy way to tell if you are looking at the last child under a parent row. One possible use for this is to print out an extra blank line once you know the print of the children is complete. The calling format is:

```
if (last_child(AA,BB,AAid))
{
  <your code here>
}
```

where

**AA** This parameter represents the name of the parent table. The actual parent row being used is the one currently pointed to by `AAcurr`.

**BB** The name of the child table under the parent.

**AAid** The name of the foreign key field that points upwards from the child to the parent. This tells the macro which pointer chain to check to see if the current BB row is the last BB child under the AA row.

The following is an example of using the `last_child` macro:

```
printf("%s %s's courses are:\n",AAcurr->fname,AAcurr->lname);
child_loop(AA,BB,BBid,AAid)
{
```

```

if (first_child(AA,BB,BBid))
{
    printf("Course\n");
    printf("-----\n");
}
printf("    %s\n",BBcurr->cname);
if (last_child(AA,BB,AAid))
    printf("\n");
}

```

## 7.5 find\_str\_loop

The find\_str\_loop macro will search the specified table for rows whose specified string field equals to the specified value. The calling format is:

```

find_str_loop("RosterView",AA,fname,value)
{
    <your code here>
}

```

where

**"RosterView"** This parameter indicates which view to apply (to determine which version number of the tables data rows).

**AA** This parameter represents the name of the table to be searched.

**fname** The name of the string field in the table to be checked .

**value** The value to be searched for.

The following is an example of using the find\_str\_loop macro:

```

printf("students with first name 'Jon' are:\n");
find_str_loop("RosterView",AA,fname,"Jon")
    printf("    %s %s\n",AAcurr->fname,AAcurr->lname);

```

**NOTE:** To search an entire table (ie, regardless of version number), use the find\_str\_loop\_all() macro. This macro does not take in a viewname.

**NOTE:** The find\_str\_loop and find\_str\_loop\_all macros can no longer be used to search for key values. Two new macros, find\_key\_loop and find\_key\_loop\_all, have been provided for this purpose.

## 7.6 find\_key\_loop

The `find_key_loop` macro will search the specified table for rows whose specified key field equals to the specified value. The calling format is:

```
find_key_loop("RosterView",AA,AAid,BBcurr->AAid)
{
    <your code here>
}
```

where

**"RosterView"** This parameter indicates which view to apply (to determine which version number of the table's data rows).

**AA** This parameter represents the name of the table to be searched.

**AAid** The name of the key field in the table to be checked.

**BBcurr->AAid** The key value to be searched for.

The following is an example of using the `find_key_loop` macro:

```
printf("Jon Anderson's classes are:\n");
find_str_loop("RosterView",AA,fname,"Jon")
    find_str_loop("RosterView",AA,lname,"Anderson")
find_key_loop("RosterView",BB,AAid,AAcurr->AAid)
    printf("    %s \n",BBcurr->cname);
```

**NOTE:** To search an entire table (ie, regardless of version number), use the `find_key_loop_all()` macro. This macro does not take in a viewname.

## 7.7 find\_int\_loop

The `find_int_loop` macro will search the specified table for rows whose

specified integer (or float) field equals the specified value. This macro is identical to the `find_str_macro`, except for the type of the field to be searched. The calling format is:

```
find_int_loop("RosterView",AA,fname,value)
{
    <your code here>
}
```

**NOTE:** Applications should avoid performing exact comparisons of floating point numbers, due to problems with the precision of the internal representation of the numbers. For example, the code:

```
x = 42.11;
y = 42.11;
if (x != y)
    printf("The impossible has happened!!!\n");
```

You may be suprised how often the 'impossible' happens!

**NOTE:** To search an entire table (ie, regardless of version number), use the `find_int_loop_all()` macro. This macro does not take in a viewname.

## 7.8 pr\_set\_int

The `pr_set_int` macro will set a schema table field of type `i2` or `i4` to a given value. The calling format is:

```
pr_set_int(tbl_elt, field, value)
```

where

**tbl\_elt** This parameter is the table element pointer created by `pr_create()`;

**field** This parameter represents the field name of the element within the table.

**value** The integer value to be assigned to the field.

The following is an example of using the `pr_set_int` macro:

```
struct AA *AA_elt;

AA_elt = pr_create(AA);
pr_set_int(AA_elt, rowdim, 5);
```

## 7.9 pr\_setflt

The `pr_setflt` macro will set a schema table field of type `f4` or `f8` to a given value. The calling format is:

```
pr_setflt(tbl_elt, field, value)
```

where

**tbl\_elt** This parameter is the table element pointer created by `pr_create()`;

**field** This parameter represents the field name of the element within the table.

**value** The float value to be assigned to the field.

The following is an example of using the `pr_set_flt` macro:

```
struct EL *EL_elt;

EL_elt = pr_create(EL);
pr_set_flt(EL_elt, value, 2.5);
```

### 7.10 `pr_set_str`

The `pr_set_str` macro will set a schema table field of type `cX` or `tX` to a given value. The calling format is:

```
pr_set_str(tbl_elt, field, value)
```

where

**tbl\_elt** This parameter is the table element pointer created by `pr_create()`;

**field** This parameter represents the field name of the element within the table.

**value** The character string value to be assigned to the field.

The following is an example of using the `pr_set_str` macro:

```
struct AA *AA_elt;

AA_elt = pr_create(AA);
pr_set_str(AA_elt, name, "MYMATRIX");
```

**NOTE:** If the length of a field value of type `cX` is less than `X`, the string is padded with spaces when the database is stored.

**NOTE:** If the length of a field value of type `cX` is longer than `X`, only the first `X` characters will be assigned and the following warning message will be displayed:

Warning: field <field name> in table <table abbreviation> is too long - truncated

## 7.11 pr\_get\_int

The `pr_get_int` macro will get the value of a schema table field type `i2` or `i4`. The calling format is:

```
pr_get_int(tbl_elt, field)
```

where

**tbl\_elt** This parameter is the table element pointer created by `pr_create()`;

**field** This parameter represents the field name of the element within the table.

The following is an example of using the `pr_get_int` macro:

```
struct AA *AA_elt;
int n_rows;

n_rows = pr_get_int(AA_elt, rowdim);
```

## 7.12 pr\_getflt

The `pr_getflt` macro will get the value of a schema table field of type `f4` or `f8`. The calling format is:

```
float pr_getflt(tbl_elt, field)
double pr_getflt(tbl_elt, field)
```

where

**tbl\_elt** This parameter is the table element pointer created by `pr_create()`;

**field** This parameter represents the field name of the element within the table.

The following is an example of using the `pr_getflt` macro on a field named 'value' in the sparse matrix element table row at `EL_elt`:

```
struct EL *EL_elt;
float elt_value;

elt_value = pr_getflt(EL_elt, value);
```

### 7.13 pr\_get\_str

The `pr_get_str` macro will get a pointer to the value of a schema table field of type `cX` or `tX`. The calling format is:

```
char * pr_get_str(tbl_elt, field)
```

where

**tbl\_elt** This parameter is the table element pointer created by `pr_create()`;

**field** This parameter represents the field name of the element within the table.

The following is an example of using the `pr_get_str` macro:

```
struct AA *AA_elt;  
char *matrix_name;  
  
matrix_name = pr_get_str(AA_elt, name);
```

### 7.14 pr\_set\_default

The `pr_set_default` macro will specify how `pr_add()` should treat an unassigned foreign key. The calling format is:

```
pr_set_default(flag)
```

where

**flag** This parameter is either 1 or 0. A one will enable default foreign key assignment, a zero will disable default foreign key assignment.

The following is an example of using the `pr_set_default` macro:

```
pr_set_default(1);
```

**NOTE:** If a table field has not been set, a default value will be used. The following default values can be assumed:

1. A value of 0 for table fields of type `i2` and `i4`.
2. A value of 0.0 for table fields of type `f4` and `f8`.
3. A value of "?" for table key fields (non foreign) of type `cX` or `tX`.
4. Primary keys are automatically generated and assigned by `pr_add( )`.
5. Foreign keys take the value of the last parent primary value added, if the default foreign key assignment has been enabled and the foreign key has not been set.

## 7.15 pr\_get\_default

The `pr_get_default` macro will return the current value of a default integer flag. The calling format is:

```
int pr_get_default()
```

The following is an example of using the `pr_get_default` macro:

```
int flag;  
  
flag = pr_get_default();
```

## 7.16 pr\_set\_key

```
pr_set_key(tbl_elt,field,value)
```

where

**tbl\_elt** This parameter is the table element pointer created by `pr_create()`;

**field** This parameter represents the field name of the element within the table.

**value** Contains the encoded version of a character key.

The following is an example of using the `pr_set_key` macro:

```
struct AA *AA_elt;  
  
AA_elt = pr_create(AA);  
pr_set_key(AA_elt, AAid, pr_get_key(AA_elt, AAid));
```

## 7.17 pr\_get\_key

```
pr_get_key(tbl_elt,field)
```

where

**tbl\_elt** This parameter is the table element pointer created by `pr_create()`;

**field** This parameter represents the field name of the element within the table.

The following is an example of using the `pr_get_key` macro:

```

struct AA *AA_elt;
int key;
key= pr_get_key(AA_elt, AAid));

```

## 7.18 "Access Macros"

### i. General Usage -

Access Macros are a special class of macros that are generated by CHGEN custom to your schema. They are used to access data fields from parent records directly. This type of access is called inheritance. For example, suppose your schema contains this parent/child relationship:

```

+-----+
| AA |
+-----+
|
|
V
+-----+
| BB |
+-----+
|
|
V
+-----+
| CC |
+-----+
|
|
V
+-----+
| DD |
+-----+

```

Now, suppose your application is currently working with a record in the DD table, and you wish to access field "zipcode" in the AA parent record that goes with this DD record. You can do this:

```

printf("zipcode=%s\n",DDcurr->CCid_pp->BBid_pp->AAid_pp->zipcode);

```

With access macros, a macro called DD\_\_AA has been created that stands for the middle portion of this long, error prone statement. Using the macro, you can simply do this:

```

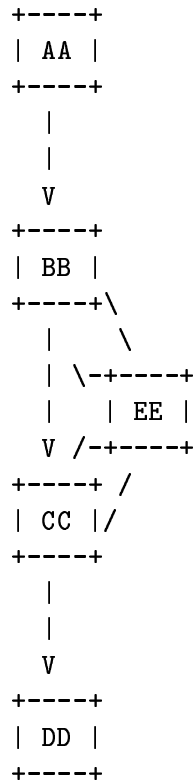
printf("zipcode=%s\n",DDcurr->DD__AA->zipcode);

```

CHGEN generates one macro for each "unique" path upwards through the schema. If a path is not unique, no macro is created. You can examine the various access macros by looking at the appropriate section of the ".h" file generated by CHGEN (using an editor, search for the string "BEGIN ACCESS MACROS"). The definitions indicate unique paths.

ii. Points of Ambiguity -

An ambiguity point occurs in the schema whenever there is more than one path to reach a derived parent. You can get around "ambiguity points" in the path by simply using multiple macros, and manually specifying what to do at the trouble spot. For example, suppose the schema is now:



The code would now become:

```
printf("zipcode=%s\n",DDcurr->DD__CC->BBid_pp->BB__AA->zipcode);
```

iii. Advantages -

Although using the macros may seem useless in this example (since the only macros used are only one "step" long), they would be handy if the path was longer. A much more important but subtle advantage is related to data independence. Suppose in the example another table, "ZZ" is inserted between tables "CC" and "DD". If the path were hard-coded in your application, you would have to change the code to accomodate the change in the data definition. However, since the macro was used, you would only have to re-compile (after re-running CHGEN to reflect the new schema). Please note that you will still have to change the application if an ambiguity point is inserted, or if a new table is inserted along a path you were forced to spell out manually.

## 7.19 User-Variable Macros

A set of macros is provided that will allow the caller to specify the table variable(s) to be used. The standard macros automatically use the built in `XXcurr` variable. These alternate macros are useful when performing a double traversal of a table (loop within a loop). The calling formats are almost the same as their standard macros, except user variables are passed in as well. The calling formats for each user-variable macro are:

```
var_table_loop(viewname,AA,uservar-AA) ...
var_table_loop_all(AA,uservar-AA) ...
var_child_loop(AA,uservar-AA,BBid,BB,uservar-BB,AAid) ...
var_first_child(AA,uservar-AA,uservar-BB,BBid) ...
var_last_child(AA,uservar-AA,uservar-BB,AAid) ...
pr_var_find(viewname,AA,uservar-AA,AAid,value) ...
var_find_str_loop(viewname,AA,uservar-AA,fname,value) ...
var_find_str_loop_all(AA,uservar-AA,fname,value) ...
var_find_int_loop(viewname,AA,uservar-AA,fname,value) ...
var_find_int_loop_all(AA,uservar-AA,fname,value) ...
```

## 8 View Definition Files

A view definition file tells the `pr_` routines the different ways users wish to view the data. Suppose for example the schema has 100 tables, but your application only needs to deal with a small subset of those tables. This subset of tables is called a view. Now suppose your application also (occasionally) needs to see the "previous version" of that same subset of tables. This might be another view. A view is defined as a list of table names, along with version numbers of each table. Since hard-coding version numbers into a view is usually not a very good practice, the view definition syntax lets you specify relation version numbers, relative to the highest existing version. Further, each view has attached to it a mode: read, update, or write. Here however, the mode applies to the entire view. Read means that your application will only be reading this data. If it does in fact output the data using `pr_dump`, all rows will still retain their version numbers (as loaded). Update mode means that the application will be changing data fields within the data (ie, non key fields). Update views will also maintain their originally loaded version number when they are output. Write mode means that a new version number should be output. The new version number is ALWAYS one version higher than the highest version in existence for that table. When specifying version numbers for tables, the special version number 0 (zero) means "highest version". A positive value implies an actual (hard-coded) version number, and a negative number implies a relative version number (based off the highest existing version number).

NOTES:

- Whitespace is ignored in the view definition file.

- Comments can occur alone on lines (using the "C" comment symbol).
- Comments cannot span multiple lines.
- Comments can occur at the end of any statement line.
- View names can be any string, but must be less than 30 characters.
- The same table CANNOT be referenced multiple times in the same view.

Consider the following view definition file for the students example:

```

/*****
/* Use the standard "C" language comment symbol for comments, but */
/* please note that comments cannot span (while open) across lines. */
/* NOTE: view names can be any string up to 30 chars long. */
*****/

define_view RosterView update /* all highest versions, for update */

view_element RosterView AA 0
view_element RosterView BB 0

define_view NewYear write /* all highest versions, for write */
/* we use this at end of semester */
view_element NewYear AA 0 /* to generate a new version. */
view_element NewYear BB 0

define_view StudentsOnly update /* student listing only, for read */

view_element StudentsOnly AA 0

define_view LastYearsRoster read /* last years roster, for read */

view_element LastYearsRoster AA -1
view_element LastYearsRoster BB -1

define_view CharterStudents read /* novelty list of first year class */

view_element CharterStudents AA 01

```

## 9 A Complete Example

This section shows a complete example program that uses the following schema file. This example assumes CHGEN has been invoked for a keysize of 8 characters.

```

*****
*      schooldb.sch      */
*****

```

```

each CC /* table CC associates courses with professors */

```

```

CCid  teacher_id      c8      1 /* primary key field */
tname teacher_name    t30     0 /* teacher's name */

```

```

oom DD /* rooms for courses */

```

```

Did   room_id        c8      1 /* primary key field */
oom   room_num       i2      0 /* room number */

```

```

oster BB /* table BB associates courses with teachers & rooms */

```

```

BBid roster_id      c8      1 /* primary (surrogate) key field */
CCid teacher_id      c8      s /* one teacher per course */
DDid where_id       c8      s /* one room per course */
cname course_name    t30     0 /* catalog name of course */

```

```

tudents AA /*1 table AA holds student records */

```

```

AAid student_id     c8      1 /* primary (surrogate) key field*/
fname first_name    t30     0 /* first name of student */
lname last_name     t30     0 /* last name of student */

```

```

nrollment EE /* enrollment relation */

```

```

EEid record_id      c8      1 /* primary key */
AAid student        c8      1 /* fkey to student */
BBid course         c8      1 /* fkey to course */

```

```

*****
*      students90.dat  */
*****

```

```

B030001 CC030001 DD030001 Calculus
B030002 CC030002 DD030003 Physics
B030003 CC030003 DD030002 Scuba
B030004 CC030004 DD030004 Typing
C030001 Prof. Johnson
C030002 Prof. Turing
C030003 Prof. Mayhem

```

```

C030004 Prof. Smith
D030001 21
D030002 25
D030003 30
D030004 40
A030001 John Doe
A030002 Susie Jones
A030003 Barney Rubble
A030004 Davie Jones
A0310000 Mr. TooBigForUs
E030001 AA030001 BB030001
E030002 AA030001 BB030003
E030003 AA030002 BB030002
E030004 AA030002 BB030004
E030005 AA030003 BB030003
E030006 AA030003 BB030004
E030007 AA030004 BB030003
E030008 AA030004 BB030002

```

```

*****/
*      students.viewdefs      */
*****/

```

```

define_view RosterView update /* all highest versions, for update */
view_element RosterView AA 0 /* Table AA is in RosterView */
view_element RosterView BB 0 /* Table BB is in RosterView */
view_element RosterView CC 0
view_element RosterView DD 0
view_element RosterView EE 0

```

```

define_view NewYear write /* write new versions at semester end */
view_element NewYear AA 0 /* To generate a new version, */
view_element NewYear BB 0 /* including tables AA and BB */
view_element NewYear CC 0
view_element NewYear DD 0
view_element NewYear EE 0

```

```

define_view StudentsOnly update /* student listing only, for read */
view_element StudentsOnly AA 0 /* Includes table AA only */

```

```

define_view LastYearsRoster read /* last year's roster, for read */
view_element LastYearsRoster AA -1 /* previous version (year) */
view_element LastYearsRoster BB -1 /* of tables AA and BB */

```

```

define_view CharterStudents read /* novelty list of first year class */
view_element CharterStudents AA 01 /* first version of table AA */

```

```

*****
*      schooldb.c      */
*****

define MAIN
include <stdio.h>
include "schooldb.h"

* CHGEN declares all tables and macros */

ain()

int i;
struct AA *AA_elt;
struct BB *BB_elt;
struct CC *CC_elt;
struct DD *DD_elt;
struct EE *EE_elt;

char tempkeya[100];
char tempkeyb[100];
char tempkeyc[100];

char* view_name;

/*****
/* Initialize the network data structures, */
/* define all views, and declare */
/* ALL datafiles for this database. */
*****/

view_name = "NewYear";

/* pr_init("students.viewdefs",
"students90.dat students95.dat");*/

pr_init("students.viewdefs","students90.dat");

/*****
/* Load the existing database. */
*****/

pr_load(view_name,"students90.dat");

```

```

        /*****
        /* Print the course list with teachers */
        /* and room numbers */
        *****/
        table_loop(view_name, BB)
    {
        printf("Course: %s\n", BBcurr->cname);
        find_key_loop(view_name, CC, CCid, BBcurr->CCid)
        {
            printf("Teacher: %s\n", CCcurr->tname);
        }
        find_key_loop(view_name, DD, DDid, BBcurr->DDid)
        {
            printf("Room: %d\n", DDcurr->room);
        }
        /* separate course with a space */
        printf("\n");
    }

        /*****
        /* Print the school roster. */
        *****/
        printf("School Roster: NumStudents = %d\n\n",
        pr_rcount(view_name,AA));

        /*****
        /* Loop over each student. */
        *****/
        table_loop(view_name,AA)
    {
        printf(" %s %s\n",AAcurr->fname,AAcurr->lname);
        /*****
        /* For each student loop over their enrollment. */
        *****/
        find_key_loop(view_name, EE, AAid, AAcurr->AAid)
        {
            find_key_loop(view_name, BB, BBid, EEcurr->BBid)
        {
            printf(" %s\n",BBcurr->cname);
        }
        }
        /* skip a line after student's last course */
        printf("\n");
    } /* end-while AAcurr */

        /*****

```

```

/* Add a new course, SoftEng, to the database. */
/* First add a new teacher and existing room, */
/* then add the course with fkeys to these */
/* fields. */
/*****

/** Add a new teacher */

CC_elt = pr_create(CC);
pr_set_str(CC_elt, tname, "Prof. Ghezzi");
pr_add(view_name, CC, CC_elt);

/** Add new course */

BB_elt = pr_create(BB);
find_int_loop(view_name, DD, room, 25) /* find room 25 pkey */
{
pr_set_key(BB_elt, CCid, pr_get_key(CC_elt, CCid)); /* set new teacher fkey */
ooooo pr_set_key(BB_elt, DDid, pr_get_key(DDcurr, DDid)); /* set fkey for room */
pr_set_str(BB_elt, cname, " SoftEng"); /* set course name */
pr_add(view_name, BB, BB_elt);
decode(tempkeya,&pr_get_key(BB_elt, BBid));
decode(tempkeyb,&pr_get_key(BB_elt, CCid));
decode(tempkeyc,&pr_get_key(BB_elt, DDid));
printf("New course row : pkey=%s, fkey= %s, fkey = %s, %s\n",
tempkeya,
tempkeyb,
tempkeyc,
pr_get_str(BB_elt, cname));
}

/*****
/* Add a new course using an existing teacher */
/* and existing room. */
/*****

BB_elt = pr_create(BB);
/* find Turing's pkey */
find_str_loop(view_name, CC, tname, "Prof. Turing")
{
find_int_loop(view_name, DD, room, 40) /* find room 40's pkey */
{
ppppp pr_set_key(BB_elt, CCid, pr_get_key(CCcurr,CCid)); /* get Turing's key & put in r
pr_set_key(BB_elt, DDid, pr_get_key(DDcurr,DDid)); /* get room 40's pkey & put in roster
pr_set_str(BB_elt, cname, " Computability");
pr_add(view_name, BB, BB_elt);
decode(tempkeya,&pr_get_key(BB_elt, BBid));

```

```

        decode(tempkeyb,&pr_get_key(BB_elt, CCid));
        decode(tempkeyc,&pr_get_key(BB_elt, DDid));
        printf("New course row : pkey=%s, fkey= %s, fkey = %s, %s\n",
tempkeya,
tempkeyb,
tempkeyc,
pr_get_str(BB_elt, cname));
    }
}

    /******
    /* Delete the student 'Davie Jones' (if found).      */
    /* Since no macro exists to perform a              */
    /* 'multi-field' find, loop over all                */
    /* Jones's. If the first name is "Davie", delete it. */
    /******
    find_str_loop(view_name,AA,lname,"Jones")
{
if (strcmp(AAcurr->fname,"Davie") == 0)

    /******
    /* Delete all of Davie's enrollement records */
    /******
    {
        find_key_loop(view_name, EE, AAid, AAcurr->AAid)
    {
printf("Deleting one of Davie's enrollment records\n");
pr_delete(EE);
    }
        printf("deleting Davie Jones\n");
        pr_delete(AA);
    }
}

    /******
    /* Delete course typing, and enrollment          */
    /* records that reference this course            */
    /******
    find_str_loop(view_name, BB, cname, "Typing") /* find pkey of typing */
{
/* find courses referencing typing */
printf("Deleting course: %s\n", BBcurr->cname);
find_key_loop(view_name, EE, BBid, BBcurr->BBid)
{
    printf("Deleting a typing record\n");
    pr_delete(EE);
}
pr_delete(BB);

```

```
}
```

```
/* Add a new student, with courses */
/* Add a new student */

AA_elt = pr_create(AA);
pr_set_str(AA_elt, fname, "Jon");
pr_set_str(AA_elt, lname, "Anderson");
pr_add(view_name, AA, AA_elt);
decode(tempkeya, &pr_get_key(AA_elt, AAid));
printf("New student added : pkey=%s, fkey= %s, fkey = %s\n",
tempkeya,
pr_get_str(AA_elt, fname),
pr_get_str(AA_elt, lname));

/* and give him a course... */
/* Note that we can find the pkey */
/* for the new student by simply looking */
/* at where AAelt points, since this always */
/* points to the last row added to the AA table. */
/* EE_elt = pr_create(EE);
pr_set_key(EE_elt, AAid, pr_get_key(AA_elt, AAid));
find_str_loop(view_name, BB, cname, "Scuba")
{
pr_set_key(EE_elt, BBid, pr_get_key(BBcurr, BBid));
pr_add(view_name, EE, EE_elt);
}
decode(tempkeya, &pr_get_key(EEelt, EEid));
decode(tempkeyb, &pr_get_key(EEelt, AAid));
decode(tempkeyc, &pr_get_key(EEelt, BBid));
printf("New enrollment : pkey=%s, fkey=%s, fkey=%s\n",
tempkeya,
tempkeyb,
tempkeyc);

/* Now give him the newly added Computability course */
/* BBelt still points to Computability (last course added) */
/* AAelt still points to Jon Anderson (last student added) */
```

```

EE_elt = pr_create(EE);
pr_set_key(EE_elt, AAid, pr_get_key(AA_elt, AAid));
pr_set_key(EE_elt, BBid, pr_get_key(BB_elt, BBid));
pr_add(view_name, EE, EE_elt);
decode(tempkeya,&pr_get_key(EE_elt, EEid));
decode(tempkeyb,&pr_get_key(AA_elt, AAid));
decode(tempkeyc,&pr_get_key(BB_elt, BBid));

printf("New enrollment row :\tkey=%s, fkey= %s, fkey = %s\n",
tempkeya,
tempkeyb,
tempkeyc);

/*****/
/* Print the new course list with */
/* teachers and room numbers */
/*****/
table_loop(view_name, BB)
{
printf("Course: %s\n", BBcurr->cname);
find_key_loop(view_name, CC, CCid, BBcurr->CCid)
{
printf("Teacher: %s\n", CCcurr->tname);
}
find_key_loop(view_name, DD, DDid, BBcurr->DDid)
{
printf("Room: %d\n", DDcurr->room);
}
/* separate course with a space */
printf("\n");
}

/*****/
/* Print the new school roster. */
/*****/
printf("School Roster: NumStudents = %d\n\n",
pr_rcount(view_name,AA));

/*****/
/* Loop over each student. */
/*****/

table_loop(view_name,AA)
{
printf(" %s %s\n",AAcurr->fname,AAcurr->lname);
/*****/
/* For each student loop over their courses.*/
/*****/

```

```

find_key_loop(view_name, EE, AAid, AAcurr->AAid)
{
    find_key_loop(view_name, BB, BBid, EEcurr->BBid)
}
printf("  %s\n",BBcurr->cname);
}
}
/* skip a line after student's last course */
printf("\n");

} /* end-while AAcurr */

/*****/
/* Use pr_find to search "Calculus" this pkey value */
/* To see the professor of Calculus and save the */
/* data in "s.dat" this file. */
/*****/

    find_str_loop(view_name, BB, cname, "Calculus")
    {
pr_find(CC, CCid, BBcurr->CCid);
if (CCcurr==NULL)
printf("The professor of 'Calculus' not found, try again.\n");
else
pr_dump_row(CC, view_name, "s.dat", 1, "w");
}

/*****/
/* To dump the course title "Calculus" which to be */
/* searched into "c.dat" this file. */
/*****/

    find_str_loop(view_name, BB, cname, "Calculus")
pr_dump_row(BB, view_name, "c.dat", 1, "a");

/*****/
/* Finally, output a new copy of the database. */
/* Use same version while we add students, etc, */
/* When the roster is finalized (end of semester), */
/* we run a program that simply loads the data */
/* using the NewYear view, and dump to a new */
/* data file, incrementing the version. */
/* In other words, save one semester per version. */
/*****/

pr_dump(view_name, "students.dat", 1, "w");
pr_stats("schooldb.stats", "write");
/* end main */

```

he output of this program is :

```
*****/  
*      students.dat      */  
*****/
```

```
CC040001 Prof. Johnson  
CC040002 Prof. Turing  
CC040003 Prof. Mayhem  
CC040004 Prof. Smith  
CC040005 Prof. Ghezzi  
DD040001      21  
DD040002      25  
DD040003      30  
DD040004      40  
BB040001 CC040001 DD040001 Calculus  
BB040002 CC040002 DD040003 Physics  
BB040003 CC040003 DD040002 Scuba  
BB040005 CC040005 DD040002 SoftEng  
BB040006 CC040002 DD040004 Computability  
AA040001 John      Doe  
AA040002 Susie     Jones  
AA040003 Barney    Rubble  
AA040005 Jon       Anderson  
EE040001 AA040001 BB040001  
EE040002 AA040001 BB040003  
EE040003 AA040002 BB040002  
EE040005 AA040003 BB040003  
EE040009 AA040005 BB040003  
EE040010 AA040005 BB040006
```

```
*****/  
*      s.dat      */  
*****/
```

```
CC040001 Prof. Johnson
```

```
*****/  
*      c.dat      */  
*****/
```

```
BB040001 CC040001 DD040001 Calculus  
BB040001 CC040001 DD040001 Calculus  
BB040001 CC040001 DD040001 Calculus
```

BB040001 CC040001 DD040001 Calculus  
BB040001 CC040001 DD040001 Calculus  
BB040001 CC040001 DD040001 Calculus  
BB040001 CC040001 DD040001 Calculus

## 10 Problems Running Version 10

Some possible problems and the corresponding error messages are listed below.

### 10.1 CHGEN-F-NAMETOOLONG

When running a schema file, this error message may appear:

```
CHGEN-F-NAMETOOLONG, table abbreviation (table) for table create  
must be 2 characters long
```

In general, this error means that your table abbreviation is more than two letters long (Remember that a table abbreviation must be two characters long when using the default keysize of 8). This particular error message indicates that your abbreviation is called “table” and the table name is “create”.

The problem is that you are running a GENDB style schema file but neglected to supply the qualifier “-gendbschema” on the command line. CHGEN v9.0 is expecting a CHGEN v8.0 schema file. Try the program again using the following sequence:

```
chgen9 -gendbschema schemafile.sch
```

This should clear this problem.

### 10.2 CHGEN-F-MISSINGTBLSTART

You may also encounter this error message while running a schema file:

```
CHGEN-F-MISSINGTBLSTART, expecting start of table 'create table' not found
```

This occurs when you use the “-gendbschema” qualifier with an improper GENDB style schema file. It is possible that you are running a CHGEN v8.0 style schema file so you should leave out the qualifier.

### 10.3 CHGEN-F-NOPATH

While running a GENDB style schema file, this error may arise:

```
CHGEN-W-NOPATHS, paths are not supported in this version of chgen.
```

Paths are supported in GENDB but not in this version of CHGEN. It is possible that you have taken a legal schema file from GENDB v1.0 to run on this version of CHGEN. This will not cause a fatal error but it should be noted that this mechanism is not supported.

## 10.4 CHGEN-F-ERRNOPARENT

`CHGEN-F-ERRNOPARENT, Could not locate parent students for relation.`

While running a GENDB style schema file, this error may occur even though the parent table has been declared in a later section of the schema file. This is known as a forward reference and is not supported by CHGEN v9.0 while using a GENDB style schema file. A CHGEN v8.0 style schema file will accept a forward reference.

To avoid this problem, simply place the parent table physically before the child table in the GENDB style schema file.

## 10.5 CHGEN-F-DUPTBLID

`CHGEN-F-DUPTBLID, duplicate field-name (AAid) found in table roster (BB)`

Several types of problems in a schema file will cause this sort of message to appear. This message indicates that two or more field names within a table have the same name. This causes an error because each line needs a unique identifier.

While using a GENDB style schema file, you may notice that no such duplicate id's occur in a table. Most likely, two or more foreign keys to the same table caused the problem. Be certain that all such keys are placed in consecutive lines in a schema file.

It is also possible that a data field uses a name that is generated as the name for the primary key or a foreign key. Avoid using names for data fields that begin with the two character abbreviation for any of the tables in the schema file.

## 10.6 No metaschema.dat

If you wish to output the values of tables TT and TA, you need to provide the qualifier “-metafile” on the command line. When running this option, you will find the \*.c and \*.h utility files as well as a file called schemaname.msdat . Remember that the name of this file is the name of the schema file with “.msdat” appended to the end.

# 11 Changes Made in Version 10-log

CHGEN version 10log provides programmers with the ability to capture the stream of operations applied to a database in a log file, and to replay this logfile. Application programming interfaces to these new features are provided through the `pr_startlog`, `pr_stoplog`, and `pr_replay` routines. These new features are generated only if the log switch is provided on the command line. The new routines

`pr_startlog`, `pr_stoplog`, and `pr_replay` will return error codes indicating distinct fault conditions, in addition to printing an error message.

The *CHGEN User's Manual* was revised to document the new functions. An appendix containing a routinefile concordance was also added.

## 12 Changes Made in Version 10

The functionality of CHGEN version 10 is to modify CHGEN 9.0 to allow the `pr_find` routine to do the binary tree search if the third character of TT's comment line is set to "1". CHGEN 10 also provides a new utility `pr_dump_row()` to dump a single row. The same utility files generated by version 9.0 are output by version 10.

## 13 Release Notes 9.1

A step up to CHGEN v9.0 was created which reads in the metadata file ( *metaschema.dat* ) and outputs the same resulting utility files. The program makes use of routines created by CHGEN v8.0 by running "metaschema.sch" through it. The `pr_load` function is used to load the input and `pr_stats` is used to output the ".stats" file and `pr_dump` is used to output the ".dat" file in addition to the resulting utilities.

Version 9.1 was created from CHGEN v8.0. None of the new options added to create CHGEN v9.0 will work with v9.1.

In addition, version 9.0 allows a different type of schema file to be input as well. A schema file similar to that accepted by GENDB can now be run by using the command line switch:

**-gendbschema**

Also, the internal tables TT (Table of Tables) and TA (Table of Attributes) can also be output to a file by using the command line switch:

**-metafile**

where "filename.dat" is the output file which is specified by the user.

As of the release date (May 18, 1994), a copy of this manual can be found in the UMASS Lowell cs system in the following directory:

`/usr/proj3/case/94s523/94sgen/base/doc`

Also a copy of the final report can be found in the same directory.

These documents were written using  $\text{\LaTeX}$ . If you wish to run  $\text{\LaTeX}$  on this document in your own directory, use the following commands:

```
latex manual
latex manual
```

The command needs to be run twice to develop the Table of Contents. This also creates the manual.dvi file.

To view the document on a workstation, after running the above commands, type:

```
xdvi manual.dvi
```

On a workstation shell (not from cs if you are at UML).

To print the document on the laser printer at UML on cs:

```
dvips manual.dvi
lpr -Plaser1 manual.ps
```

This method is correct as of May 18, 1994 but may have since been altered. If you are running this on a different system, please refer to your system user's guide.

If you can not find these files on the cs filesystem, Contact Dr. Robert Lechner for details.

## 14 Release Notes 8.0

The following list briefly describes the changes and new features supported by version 8.0 of chgen, broken into major categories.

### 14.1 Running CHGEN

- No new command options have been added, since version 8 did not add any new functional capabilities, although two new macros, `pr_set_key` and `pr_get_key` were added. Its purpose is to merge the changes of version 7.0 and `pgen` and eliminate as many bugs as possible. As with all new CHGEN releases, the executable is now located in a version 8 directory. If you have a symbol defined in your login to run `chgen`, you may want to change it.
- CHGEN now represents all keys internally (in memory) as unsigned integers. This will improve the efficiency of the `chgen` generated code and will allow applications using `chgen` code to run using less memory and with more speed.
- The key representation used by CHGEN has been enlarged. CHGEN now operates on one of two defined key sizes. Either the original key size of 8 characters (2 for table abbreviation, 2 for version number, and 4 for row), or the new keysize of 12 characters (4 for table abbreviation, 3 for version number, and 5 for row) may be used. The keysize can be specified to CHGEN with the `-keysize=` command line parameter. Valid command line values are `-keysize=8` and `-keysize=12`. If no parameter specifying keysize is given, a default of 8 characters is assigned.
- Using the expanded keysize of 12 characters, the maximum row number allowed is now 16382, and the maximum version number allowed is 255.

- The CHGEN output code now compiles without compiler warnings.
- A bug relating to the view names (defined in the view definition file) not being cleared by `pr_init` has been fixed. Previously, it was not possible to do a `pr_init`, `pr_load`, `pr_dump`, `pr_free`, `pr_init` sequence, because the view names were still in memory and this was erroneously detected by `pr_init` as a duplicate view name. The bug has been fixed and `pr_init` now resets the list of view names.
- CHGEN will now assume a schema filename extension of `.sch` if the specified file has no extension. See section 3.2 for details.
- CHGEN will now correctly locate the schema file if a directory specification is provided as part of the schema filename. The output files will still be placed in the users current directory however. See section 3.2 for details.

## 14.2 Data File Changes

No changes, however, both the schema file and view definition file must contain table abbreviation sizes that correspond to the keysize parameter given to CHGEN. Schema file should use data type 'c8' to identify all keys, regardless of actual external size. Data files used by application code must also contain keys that correspond to the keysize used when CHGEN was executed.

## 14.3 Programming Interface

- Two new macros have been added for version 8, `pr_set_key` and `pr_get_key`. `pr_set_key` is used by the applications programmer to set foreign keys in tables created by the network database (primary keys are set by `pr_create`). `pr_get_key` returns an integer key.
- (From version 7.0)
- The `find_str_loop` and `find_str_loop_all` macros can no longer be used to loop on key values (primary or foreign). Two new macros that are identical in format to the old ones have been created: `find_key_loop` and `find_key_loop_all`. The only difference is that the new macros use a new `key_compare` function to compare key values.
- Temporary key values can not always be declared to be of size 9 (as is currently done), since a key can be 12 characters long. To ensure that the proper size is always allocated, temporary key values can use a new defined `hcg_key`. To declare a temporary key, the code should look like:

```
hcg_key tempkey;
```

Since all keys are internally represented as integers, any references in application code to these keys will have to change. Two new utility functions have been created to allow the user to decode a key value to obtain the character equivalent:

```
int decode(hcg_key *key_char, hcg_key *key)
```

This function will decode the key pointed to by `key` and place the character equivalent in `key_char`. The function returns a status of 1 if successful, or -1 if the key is not valid.

```
hcg_key * decode_retstr(hcg_key *key)
```

This function will decode the key pointed to by key and return the character equivalent in a static string. Please note that this function DOES NOT allocate memory and the string should be copied into a user variable immediately. This function is useful for displaying the character equivalent of a string but note that several calls to decode\_retstr CAN NOT be made within the same printf statement.

- The pr\_gen\_pkey function is unchanged. However, it has been moved to gen\_pr\_load as a first step in a future conversion to C++. All the example programs use the new macros introduced by pgen instead of calling pr\_gen\_pkey. Since pr\_add changed from :

```
pr_add(view_name, temp_buffer);
```

to :

```
pr_add(view_name, tbl_abbrev, tbl_elt);
```

the old method of adding a table element with sprintf no longer works. For example, assuming you wanted to add a new row to table BB, and table BB contains a foreign key to table AA (pointed to by AAcurr-*j*, and a course name, the code to generate a new BB row would look like:

```
BB_elt = pr_create(BB); pr_set_key(BB_elt, AAid, pr_get_key(AA_elt, AAid)); pr_set_str(BB_elt, cname, "Physics"); pr_add(viewname, BB, BB_elt);
```

- If, in the above example, a student (table AA) had just been created, and the student's pkey had been saved in variable student\_pkey, then the call to pr\_get\_key could be replaced with a reference to student\_pkey instead. This method will only work if all the foreign keys referenced by the new row have just been created (and thus are still available in temporary character keys).

- (From pgen)

- When adding a record to a database, you no longer need to build a buffer using sprintf. A new set of macros are provided to assign values to a field in the table.

pr\_create : creates a table element

pr\_set\_str\_loop : sets a string field

pr\_set\_int\_loop : sets a integer field

pr\_set\_ft : sets a floating point field

pr\_get\_str\_loop : gets a string field

pr\_get\_int : gets a integer field

pr\_get\_ft : gets a floating point field

pr\_set\_default : sets a default field

pr\_get\_default : gets a default field

Inclusion of these macros is not optional. If you do not need them, application size will not be affected (included macros that are not used do not get in the way). You could also simply delete them from the generated .h file if you wish.

The input parameters for pr\_add have been modified to be the viewname, table abbreviation, and the table element pointer.

## 15 Release Notes (from version 6.0)

The following list briefly describes the changes and new features supported by versions of chgen starting at 6.0, broken into major categories.

### 15.1 Running CHGEN

- New command options have been added, but are not necessary for default operation. As with all new CHGEN releases, the executable is now located in a version 8 directory. If you have a symbol defined in your login to run chgen, you may want to change it.
- CHGEN will now assume a schema filename extension of .sch if the specified file has no extension. See section 3.2 for details.
- CHGEN will now correctly locate the schema file if a directory specification is provided as part of the schema filename. The output files will still be placed in the users current directory however. See section 3.2 for details.

### 15.2 Data File Changes

No changes, however, a new file called a view definition file MUST be provided.

### 15.3 Programming Interface

- When using the last\_child macro, you must now pass the childs foreign key to the parent, rather than the parents foreign key to the child, as the third parameter. For example, if CC is a child of BB, and you want to know if CCcurr is the last child of BB, you must now say:  
if (last\_child(BB,CC,BBid))  
rather than :  
if (last\_child(BB,CC,CCid))  
Note that the schema itself does not define "parent foreign keys to children" explicitly. CHGEN generates pointers down to children in the form of XXid\_fcp and XXif\_bcp fields. This change was necessary to remove generic dependencies on back-pointers within CHGEN. See section 8.4 for details.
- Backward pointers are now optional on a per-foreign-key basis. By using the new value "-1" instead of "1" when specifying a foreign key in the schema file, back-pointers along that relationship will not be generated. This feature will help to reduce the memory and code size requirements of CHGEN applications. See section 3.1 for details.
- Backward pointers can be disabled for the entire schema by running CHGEN with the new -nobp flag. The qualifier is intention-ally not called -nobcp, which would imply only \_bcp (back child pointer) fields are suppressed. In fact, both \_bcp and \_bpb fields are suppressed. See section 3.6 for details.

- Support for ANSI "C" is now provided. By invoking CHGEN with the new -ansi qualifier, .h files will compile under ANSI "C" compilers such as GCC and GPP. Currently, the only difference is that the identifier concatenation operator is ## under ANSI "C", and is /\*\*/ (ie, empty comment) for K&R "C". See section 3.6 for details.
- A new command line qualifier, -validate, will cause CHGEN to only validate the schema file. It will not generate any output files. See section 3.6 for details.
- A new command line qualifier, -noforward, will cause CHGEN to generate macros that assume there are no forward references in data files (ie, child row loaded before parent). This feature will help to reduce the memory and code size requirements of CHGEN applications, and significantly improve the speed of pr\_load. See section 3.6 for details.
- A new command line qualifier, -quiet, will cause CHGEN to NOT report informational and warning statements when compiling a schema. See section 3.6 for details.
- Comments have been added to CHGEN output file, particularly the .h file. Reading some of these comments should help programmers understand how CHGEN works, and may provide insight into some obscure but powerful "C" programming techniques.
- A new set of macros have been created that are similar to existing macros, but these allow the user to specify his/her own table variables. The new macros are :
  - pr\_var\_find : similar to pr\_find
  - var\_find\_str\_loop : similar to find\_str\_loop
  - var\_find\_int\_loop : similar to find\_int\_loop
  - var\_child\_loop : similar to child\_loop
  - var\_find\_str\_loop : similar to find\_str\_loop
  - var\_table\_loop : similar to table\_loop
  - var\_first\_child : similar to first\_child
  - var\_last\_child : similar to last\_child

Inclusion of these macros is not optional. If you do not need them, application size will not be affected (included macros that are not used do not get in the way). You could also simply delete them from the generated .h file if you wish.
- A new strategy for managing versions and views has been implemented. This will require several changes to applications, mostly minor however. Please review your code with sections 9 and 10 of this manual. In addition, several new macros have been created, differentiating between view-traversal and entire-table traversal, etc...
- A new command line qualifier, -maxviews=n, will cause CHGEN to generate code that allows up to "n" views. Without this qualifier, the default of 10 views will be assumed. See section 3.6 for details.
- A new command line qualifier, -noorder, will cause CHGEN to output code that does not bother to insert rows into tables in primary-key sorted order, AND not to bother linking children into chains in data-load order. If applications do not care about ordering, this qualifier can lead to very large code size reductions. See section 3.6 for details.
- A new source code module, pr\_stats.c is output by CHGEN. This routine will print some useful statistics about the schema, views, and loaded data to the specified file. This output is in a human-readable form (more like a report than a table). A future parameter to this routine will force the output of a TS table (table statistics) output. See section 6.10 for details.

## 15.4 Miscellaneous

- Removed the use of the C-library function `memset()` from internal CHGEN code. This was done since `memset()` is not supported on some versions of unix (hawk being one of them). This is simply one more step toward making CHGEN portable.

## 16 CHGEN 10log

### 16.1 Where to find CHGEN v10log

The source files are in:

```
/usr/proj3/case/96s523/gen/mmurphy/genv10\_log/src
```

The executable is in:

```
/usr/proj3/case/96s523/gen/mmurphy/genv10\_log/executables/alpha/chgen10
```

### 16.2 Running CHGEN 10log

Simply type the name of the executable and then the input file along with any command line switches (log if you want logging support). If you would like to search by binary tree, you Must set the third character of the TT's comment line. Otherwise, it will use sequential search.

### 16.3 Output of CHGEN 10log

The output will be the following utility files

```
pr_delet.c
pr_dump.c
pr_free.c
pr_load.c
pr_stats.c
    pr_log.c (if -log switch used)
```

and

```
<schemaname>.h
```

## 17 CHGEN 10

### 17.1 Where to find CHGEN v9.1

The source files are in:

```
/usr/proj3/case/95s523/genv10/base/src/CHGEN10
```

The executable is in:

```
/usr/proj3/case/95s523/genv10/base/executables/CHGEN10
```

## 17.2 Running CHGEN 10

Simply type the name of the executable and then the input file along with any command line switches. If you would like to search by binary tree, you must set the third character of the TT's comment line. Otherwise, it will use sequential search.

## 17.3 Output of CHGEN 10

The output will be the following utility files

```
pr_delet.c  
pr_dump.c  
pr_free.c  
pr_load.c  
pr_stats.c
```

and

```
<schemaname>.h
```

# 18 Appendices

## 18.1 Appendix A: entry-point file cross-reference

The following table shows what file contains each generated routine. It is meant as an aid to application programmer's using CHGEN output. It was generated with the aid of the standard UNIX ctags command, and is accurate as of 14 May, 1996.

abb_decode	pr_load.c
abb_encode	pr_load.c
abbr_lut_create	pr_load.c
alloc_element	pr_load.c
btree_create	pr_load.c
btree_destroy	pr_load.c
btree_display_nodes	pr_load.c
btree_does_node_exist	pr_load.c
btree_initialized	pr_load.c
btree_insert_node	pr_load.c
btree_print_node	pr_load.c
btree_wt	pr_load.c
clear_mem	pr_load.c
convert2temp	pr_log.c
decode	pr_load.c
decode_retstr	pr_load.c
del_row	pr_delete.c
del_tempfiles	pr_log.c
dump_row	pr_dump.c
dup_row_warning	pr_load.c
encode	pr_load.c
find_tbl_idx	pr_load.c
find_view_idx	pr_load.c
forward_ref_warning	pr_load.c
free_table	pr_free.c
func	pr_log.c
gen_pr_add	pr_load.c
get_abbr	pr_load.c
get_delta_tee	pr_log.c
get_row	pr_load.c
get_version	pr_load.c
hcg_parse	pr_load.c
hcg_read_next	pr_load.c
hcg_update_version	pr_load.c
if	pr_dump.c
insert_element	pr_load.c
is_null_key	pr_load.c
is_string_null	pr_load.c
key_compare	pr_load.c
link_child_1	pr_load.c
link_child_bp_m	pr_load.c
link_child_nobp_m	pr_load.c
link_parent_1	pr_load.c
link_parent_bp_m	pr_load.c
link_parent_nobp_m	pr_load.c
load_data	pr_load.c
log_do_add	pr_log.c
log_do_delete	pr_log.c
log_parselogdata	pr_log.c
log_sleep	pr_log.c
logstr	pr_log.c
logwait	pr_log.c
lut_delete_element	pr_load.c
lut_display	pr_load.c
lut_get_name	pr_load.c
lut_init	pr_load.c
lut_insert_element	pr_load.c

## 19 References

Lechner, Dr. Robert J. & Smith, Craig & Smith, Stephen, Data Modeling Support For C Programmers, Computer Science Department at University of Massachusetts at Lowell, October 2, 1992.

Moretti, Michael & Powell, Justin & Senevirante, Surekha, CHGEN Users Manual Version 8.0, Computer Science Department at University of Massachusetts at Lowell, August 31, 1993.

Moretti, Michael & Powell, Justin & Senevirante, Surekha, Final Report GEN Team, Computer Science Department at University of Massachusetts Lowell, August 18, 1993.

Smith, Stephen C. & Smith, Craig E., GENDB- A Data-Modeling Tool For Database Application Development, Computer Science Department at University of Massachusetts at Lowell, May 1, 1992. Revised by Dr. Robert J. Lechner February 4, 1994.

Prof. Lechner, Hitoshi Sano, Michael Cook, & Jung-Rim Hyun, CHGEN 9.0 User's Manual, & CHGEN 9.0 Final Report, Computer Science Department at University of Massachusetts Lowell, May 18, 1994.

Denise Nelson, Bill Rideout, & Mike Murphy, CHGEN10LOG Final Report, Computer Science Department at University of Massachusetts at Lowell, May 13, 1996.