# PROGRAM REFERENCE FOR KIC

by

G. C. Billingsley

Memorandum No. UCB/ERL M83/62

3 October 1983

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

**Giles C. Billingsley**

Author

# Program Reference for KIC

Title

RESEARCH PROJECT

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, in partial satisfaction of the requirements for the degree of Master of Science, Plan II.

Approval for the Report and Comprehensive Examination:

Committee: Richard Newton Research Advisor

8/29/83 Date

8/29/83 Date

# PROGRAM REFERENCE FOR KIC

*Giles C. Billingsley*

Electronics Research Laboratory

Electrical Engineering and Computer Science Department

University of California

Berkeley, California 94720

*ABSTRACT*

The internal structure of the KIC interactive, color graphics editor is explained in this report. The database, user interface, and system dependencies are examined, and information is presented to assist the programmer who would wish to extend the KIC program.

August 28, 1983

# Acknowledgements

I would like to thank all those who have contributed to the development of the KIC editor. The history of KIC and all related programs spans several years, and appropriately there are many persons to be acknowledged for their kind help and support.

The invaluable advice and encouragement of Professor Richard Newton is greatly appreciated. Professor Newton has guided the development of the KIC editor from the time that the project was begun by Kenneth H. Keller in the Fall of 1980, and he has provided excellent opportunities for both the KIC program and myself. The opportunities for me include a Research Assistantship funded in part by Tektronix Inc., National Semiconductor Corp., and Digital Equipment Corp., the opportunity to develop the KIC editor with an excellent user community including Berkeley graduate students in the Integrated Circuits Group and designers from National Semiconductor, Evans and Sutherland Corp., CSIRO Australia, and Tektronix, and the opportunity to be part of perhaps the world's most prestigious university CAD effort in integrated circuits.

The support and professional experiences provided by Professor D. O. Pederson are also greatly appreciated. Professor Pederson, along with Dr. Stewart Taylor of Tektronix, convinced me to enter the graduate study program at Berkeley and has continually helped to make my studies at Berkeley productive and beneficial.

I am very grateful to those persons who have provided assistance and software during the development of the KIC program. These persons include Kenneth H. Keller who initially designed and wrote KIC and the CD database package, Peter P. Moore who developed the nmalloc memory management package, Chris Wilson of CSIRO who wrote the initial version of geometry stretching procedures, Peter Harris of Jupiter Systems Inc. for firmware support on the AED 512, Professor J. K. Ousterhout who provided the UNIX directory search path procedures, and Professor Carlo Sequin for his early interest in the KIC program and his helpful suggestions for the user interface. Also, Mark Bales implemented the first version of the MFB graphics package.

I also gratefully acknowledge the helpful support from the early industrial users of KIC. Dr. J. Craig Mudge and Chris Wilson of the VLSI design group at the Division of Computing Research, CSIRO, Australia, provided both ideas and software for the geometry stretching commands in KIC. J. Halverson and R. Haslan of Evans and Sutherland have provided help for the development and improvement of KIC under VMS. S. Sriram, F. Taku, and S. Carcia of the Engineering Systems Group of Digital Equipment Corp. and A. Hanover of the Methodology and Advanced Development Group of Digital Equipment Corp. have also given considerable support to the development of KIC on VMS. Jim Solomon and Joe Santos of the Speech Design Group of National Semiconductor, where I worked on KIC as a summer employee during the Summer of 1982, were the first major industrial users of KIC in the Santa Clara area, and I am very grateful for their kind support. The work of Ed Gould, who helped to improve the user interface of the CIF conversion programs and who wrote a new .KIC parser, is appreciated. Tektronix has been generously supportive of my work by providing graphics and computing equipment, financial support, and the opportunity to share ideas with their excellent CAD effort; in particular, I would like to acknowledge the assistance and advice of Dr. E. Cohen, J. Crawford, and S. Potter of the Tektronix MCE CAD group.

I would also like to thank Dr. I. Getreu of Tektronix for his support and encouragement throughout the course of this work.

The most important acknowledgements, the personal ones, are saved for last. To my parents in Oregon, Virgil and Rosemary, I am deeply thankful for their their constant, loving support and infinite patience during my graduate years. And to my Uncle Jess and Aunt Betty Billingsley I am especially grateful for their understanding and for making certain that this over-worked Berkeley graduate student enjoyed at least one good, home-cooked meal every month!

Finally, to the friends that I have found at Berkeley who have given me helpful criticism and assistance throughout the course of my research project. I thankfully acknowledge the generous assistance of Clem Cole, Mark Hofmann, Ken Keller, Jim Kleckner, Grace Mah, Peter Moore, Tom Quarles, and Rick Spickelmier.

## Table of Contents

**Chapter 1: Introduction** ........................................ 1

**Chapter 2: The CD Database** ........................................ 4

2.1. Introduction ........................................ 4

2.2. CD Descriptor Types ........................................ 7

2.2.1. The Symbol Descriptor ........................................ 7

2.2.2. The Master-List Descriptor ........................................ 9

2.2.3. The Symbol Table Descriptor ........................................ 10

2.2.4. The Object Descriptor ........................................ 11

2.2.5. The Generator Descriptor ........................................ 12

2.2.6. The Transform Descriptor ........................................ 13

2.2.7. The Property List Descriptor ........................................ 14

2.2.8. The CD Layer Descriptor ........................................ 15

2.2.9. The Path Descriptor ........................................ 16

2.3. Storage Bins ........................................ 17

2.4. CD Procedures ........................................ 20

2.4.1. CD Initialization ........................................ 20

2.4.2. Error Handling in CD ........................................ 20

2.4.3. CD Symbol Management ........................................ 21

2.4.4. CD Object Creation Routines ........................................ 24

2.4.5. CD Object Generator ........................................ 29

2.4.6. Accessing Objects in CD ........................................ 30

2.4.7. Object Deletion in CD ........ 33

2.4.8. CD Information Routines ........ 33

2.4.9. CD Integrity ........ 35

2.5. Two Dimensional Transformation Package ........ 37

2.5.1. Initialization ........ 37

2.5.2. The Current Transformation ........ 37

2.5.3. The Transformation Stack ........ 38

2.5.4. The Instance Transformation ........ 40

2.5.5. Inverse of the Current Transformation ........ 40

2.6. Traversing a Symbol Hierarchy with CD ........ 42

2.7. Translation Routines ........ 45

2.8. The CD Parameters Descriptor ........ 48

**Chapter 3: The Fast CIF Parser** ........ 50

3.1. Action Routines ........ 51

**Chapter 4: The KIC User Interface** ........ 56

4.1. Window and Viewport Management ........ 56

4.1.1. Text Viewports ........ 57

4.1.1.1. Layer Menu Viewport ........ 58

4.1.1.2. Parameter Viewport ........ 59

4.1.1.3. Prompt Viewport ........ 59

4.1.1.4. Command Menu Viewport ........ 60

4.1.2. Layout Viewports ........ 61

4.1.3. Clipping ........ 63

4.1.4. Window/Viewport Transformations ........ 64

4.2. KIC Data Structures ........ 67

4.2.1. The Area Descriptor ........ 67

4.2.2. The Window Stack Descriptor ........ 68

4.2.3. The KIC Layer Table Descriptor ........ 68

4.2.4. The Cursor Descriptor ........ 71

4.2.5. The KIC Selection Queue ........ 72

4.2.6. The Context Descriptor ........ 72

4.3. The KIC Parameters Structure ........ 75

4.4. KIC Command Menus ........ 86

4.5. The Pointing Device ........ 92

4.6. The Frame Buffer Interface ........ 98

4.6.1. The Frame Buffer Descriptor ........ 98

4.6.2. The Frame Buffer Routines ........ 101

4.7. Geometry Display Routines ........ 113

4.7.1. Redisplay ........ 114

4.8. Geometry Input Routines ........ 117

4.9. Geometry and Symbol Modification Procedures ........ 120

**Chapter 5: System Dependencies** ........ 124

5.1. Terminal I/O Dependencies ........ 124

5.2. The Directory Search Path ........ 124

5.3. Memory Management ........ 125

5.4. The System Interface ........ 126

**Appendix A: A Catalog of All Routines and Macros** ........ A.1

**Appendix B: The CD Programmers's Manual** ........ B.1

Appendix C: The KIC Tutorial User's Guide ........................................................ C.1

Appendix D: The MFB Programmers's Manual ................................................ D.1

Appendix E: The MFBCAP Programmer's Manual ........................................... E.1

Appendix F: KIC and Related Manual Pages ...................................................... F.1

References .................................................................................................. R.1

# Chapter 1

# Introduction

KIC [1,2] is an interactive, two-dimensional, color graphics editor intended primarily for the mask level design of integrated circuits. It is written in the C programming language and runs in either a *UNIX*[1] or *VMS*[2] environment.

KIC has been designed as a powerful, inexpensive, user-friendly graphics editor that will run on most low to medium performance graphics terminals. Data that is generated by KIC can be represented by an intermediate graphic description language, such as CIF (Caltech Intermediate Form) or Calma *STREAM*,[3] which permits the data to be easily transported to other layout systems. Also, the geometric database used by KIC can be used to interface to other tools, such as a layout rules checking program [8].

The internal structure of the KIC program is described in detail in this report. The reader must be familiar with the C programming language as defined in [3] or [9]. Other KIC documentation includes a database manual and a user's tutorial, both included as appendices to this report. A programmer should be thoroughly familiar with this report and these two appended manuals before attempting an addition or enhancement to the KIC program.

KIC can be viewed as four major subsystems, as illustrated in Figure 1.

KIC uses the *CD* (CIF Database) relation database system that manages a set of files. Each file contains the definition of a layout-cell that is stored as an ASCII CIF symbol description [5]. A cell definition can contain instances of other

[1] UNIX is a trademark of Bell Laboratories.
[2] VMS is a trademark of Digital Equipment Corp.
[3] STREAM is a trademark of Calma, Inc.

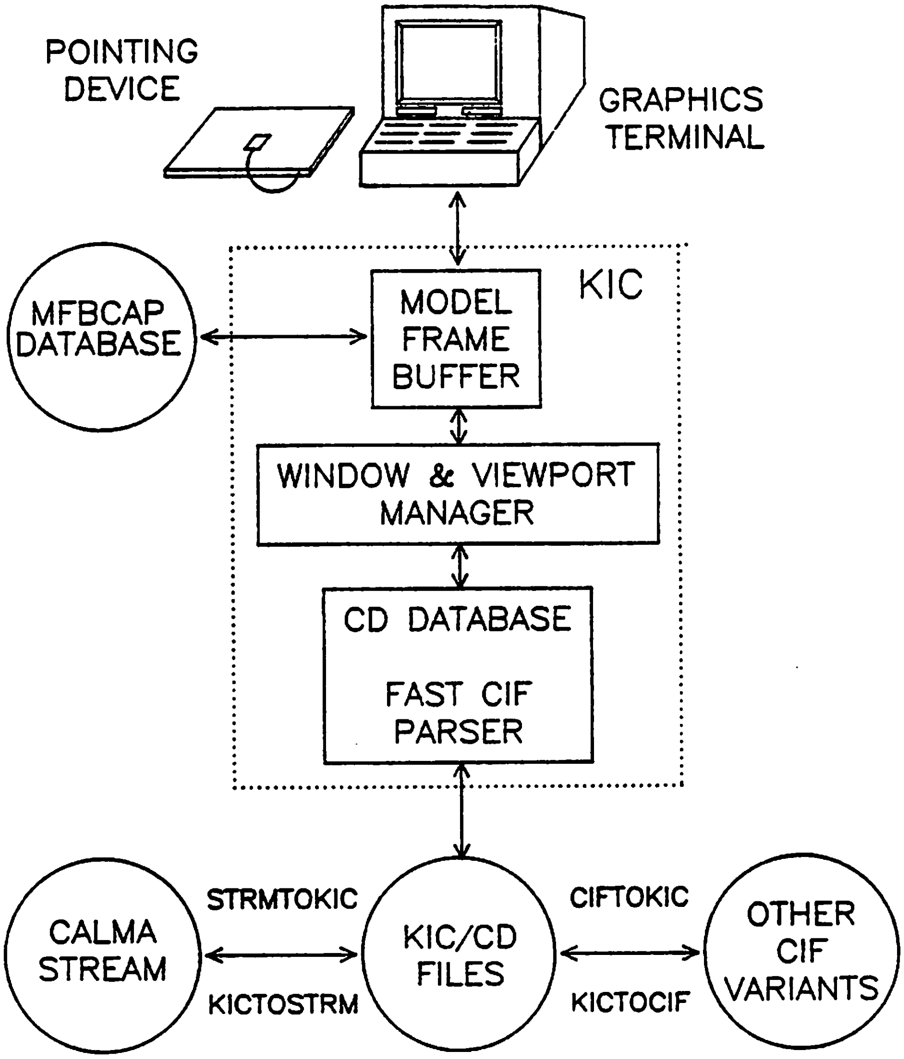

**Figure 1. The KIC system.**

cells, and thus the database supports hierarchical layout descriptions. Other layout formats, such as Calma STREAM format, can be obtained from CD translation routines [11]. The CD database package reads these layout descriptions from disc files and transforms them into virtual-memory data structures. This package is described in detail in Chapter 2. A major part of this process concerns the parsing of CIF data from the resident disk files. A fast CIF parser has been developed for this purpose and is described in Chapter 3.

KIC is designed to run on a wide range of raster graphics terminals or frame buffers and indeed has been used with several devices including the AED 512 and

767, the Tektronix 4113 and 4105, the HP2648A, the Metheus Omega-400, and the Masscomp MC500 series computer. KIC uses the notion of an *ideal* color graphics frame buffer, and a separate package of routines maps commands for this *ideal* frame buffer to the real device. This package can be a set of hard-coded routines for the particular terminal, or the *Model Frame Buffer* (MFB) package can be used. MFB is a terminal independent graphics package that uses an ASCII database file (MFBCAP) to represent the frame buffer characteristics in an extended UNIX termcap format; see *termcap(5)* and *curses(3)* in the BSD UNIX programmer's manual. MFBCAP and the associated MFB routines are documented in Appendix D and E. A description of how to interface KIC to a new graphics terminal is contained in Section 4.6.

# Chapter 2

# The CD Database

## 2.1. Introduction

CD (CIF Database) is a package of C procedures for managing CIF databases at an object level; geometric objects are modeled as CIF geometries and grouped as CIF symbols. For a description of the CIF language, see [4] or [5]. The KIC program is only one application of the CD database package, other applications including language conversion programs between KIC/CD format, CIF, and Calma STREAM.

Because the data model of CD is CIF, the CD database inherits both the advantages and limitations of the CIF language. As an example, one important feature of CIF is that it is hierarchical. However, called and placed symbols can not be magnified or arrayed without adopting nonstandard extensions of the language. Such limitations are described in the various sections of this chapter.

The reader should be aware of the following CD terminology: a ***master*** refers to the definition of a symbol, and an ***instance*** refers to a call and placement of a symbol within a ***master***. To explain the phrase ***traversing a symbol***, think of the symbol as a tree structure where each element is an instance that is linked to its master. A procedure that traverses a symbol would visit each branch of the symbol-instance tree structure. Finally, a ***lambda*** coordinate is one that is within the coarse resolution of CD; for example, if the CD database unit is one one-hundredth of a micron and the coarse resolution of CD is one hundred database units, then the minimum lambda value is one micron. This coarse resolution allows coordinates to be represented as integers while providing a finer resolution when smooth contours are required.

When a symbol is opened via the *CDOpen()* routine, it is mapped into main memory from local or library files, each storing one CIF symbol definition. Also, all symbols that are called by the symbol are read into main memory. A symbol that has been opened is referenced by a ***symbol descriptor*** defined in the next section. To reflect at its secondary storage site the changes to a symbol that has been opened, a program invokes the *CDUpdate()* routine. To remove open symbol unknown from CD (i.e., to remove it from main memory), a program invokes the *CDClose()* routine.

Geometries or objects are collectively organized in a symbol or cell. The types of valid objects within a symbol are boxes, polygons, wires, round flashes, and symbol calls or instance arrays. For each object there is a procedure that creates the object in a particular symbol on a particular layer. These data insertion routines are *CDMakePolygon()*, *CDMakeWire()*, *CDMakeBox()*, and *CDMakeRoundFlash()*. The CIF *call* has been extended to handle instance arrays. To create an instance array, a program invokes the *CDBeginMakeCall()*, *CDT()*, and *CDEndMakeCall()* procedures. All object creation procedures return a pointer to an ***object descriptor*** that has the purpose of referencing the object in the database. The *CDDelete()* procedure removes an object from a master cell.

There are several routines for acquiring or storing information that is specific to a particular object or symbol. The ***bounding box*** of an object can be accessed via the *CDBB()* routine. Associated with each object is an integer information field that can be accessed by the *CDSetInfo()* and *CDInfo()* routines for extending that object's description. In addition to the integer information field, each object can have a linked-list of property strings that are accessed via the *CDAddProperty()* and *CDProperty()* procedures.

CD uses a two dimensional, rectilinear or "Manhattan" transformation package that can be used by any CD application program. The transformation package is described in Section 2.5. With the transform package, a program can define a current transformation composed of translations, reflections, and rotations, obtain the transformation and inverse transformation of coordinates, and manage several transformations with a transformation stack.

Traversing the contents of a master is performed with a ***object generator loop***. In the context of CD, an object generator is a set of procedures that allow the program to sequentially access objects on a particular layer that intersect a particular area. To begin a generator, a program would invoke the *CDInitGen()* routine with an area of interest and a specific layer as parameters. The procedure will return a pointer to a ***generator descriptor***, defined in the next section. Every invocation of the *CDGen()* procedure will then return a pointer to an object descriptor whose bounding box, transformed by the current transformation, intersects the area and lies on the particular layer. When *CDGen()* has returned all objects that qualify, the procedure returns a null pointer to the object descriptor.

The *CDType()* procedure returns the type of an object descriptor, (e.g., box, polygon, etc.) and can be used to dispatch to a type-specific procedure for manipulating the object.

CD provides several routines for translating to or from CIF. The *CDTo()* routine translates CIF directly into CD format. *CDFrom()* translates a CD symbol hierarchy into a CIF file. Also, the *CDParseCIF()* procedure will build the CD database from a CIF file rather than from a directory of symbol files. This latter procedure is useful for translating CIF into other layout languages such as Calma STREAM. These routines are explained in Section 2.7.

### 2.2. CD Descriptor Types

There are several descriptor types defined as structures in the cd.h file.

| **descriptor type** | **structure name** |
|---|---|
| symbol | s |
| symbol table | bu |
| master-list | m |
| object | o |
| label | la |
| polygon | po |
| round flash | r |
| wire | w |
| call | c |
| transform | t |
| generator | g |
| property | prpty |
| layer | l |
| path | p |

The following sections describe the various descriptors in greater detail.

### 2.2.1. The Symbol Descriptor

The symbol descriptor is defined in the *cd.h* file as follows:

```
/*
 * Symbol desc.
 */
struct s {
     int sLeft, sBottom, sRight, sTop;
     int sBBValid;
     int sA, sB;
     char *sName;
     short sInfo;
     struct o ***sBin[CDNUMLAYERS+1];
     struct m *sMasterList;
     struct prpty *sPrptyList;
     };
```

A symbol descriptor is associated with each symbol that resides in the database and is necessary for accessing the objects contained in the respective symbol. This descriptor is allocated and initialized by the *CDOpen()* routine only and released by the *CDClose()* routine. All current symbol descriptors are referenced in the CD symbol table; the symbol table descriptor is described in the next section.

The *sName* member is a pointer to a null-terminated character string containing the name of the respective symbol. The bounding box of the symbol in coordinates is given by *sLeft*, *sBottom*, *sRight*, and *sTop*. Because the bounding box of the symbol depends on the objects contained in the symbol, there is a time (for example, when the CIF is being parsed or the symbol is being opened) when the bounding box is not valid. This condition is flagged by the *sBBValid* boolean member of the symbol descriptor.

The *sA* and *sB* members define the magnification factor of the symbol. KIC and the CD database do not permit symbol magnification, and therefore these members are always set to unity. If such a feature is added to KIC, it should be done at the symbol level rather than at the object level to be consistent with the CIF data model. For example, the magnification factor of any geometry or instance in a symbol would always be unity. However, the magnification factor of a symbol definition that is called by another would not necessarily have to be unity. This symbol magnification factor would be defined by a ratio of two integers in the *DS* command in the respective symbol file and these two integers would become the *sA* and *sB* members in the symbol descriptor when the symbol is opened by CD.

The objects contained in a symbol are organized in a storage bin structure that is explained in greater detail in Section 2.3. In brief, the storage bins are a three dimension array of doubly linked object descriptor lists; each object list is indexed in the array of bins according to its layer and the lower, left coordinate of its bounding box. *sBin* is a pointer to the symbols storage bins. The bin structure is declared as a triple star ( ***sBin[CDNUMLAYERS+1] ) because the bins are allocated on demand for each layer with layer zero being the instance layer.

*sInfo* is the integer information field of the symbol and is initialized to zero, and *sPrptyList* is a pointer to the symbol property list. If the *sPrptyList* pointer is NULL, the property list is empty. *sMasterList* is a pointer to the symbol's master-list that is explained in greater detail below.

### 2.2.2. The Master-List Descriptor

The master-list descriptor is defined in the *cd.h* file as follows:

```
/*
 * Master-list desc.
 */
struct m {
     int mReferenceCount;
     int mLeft, mBottom, mRight, mTop;
     char *mName;
     struct m *mPred, *mSucc;
     };
```

Every symbol in CD has one master-list. The master-list is a doubly linked-list of structures containing one link for each symbol that is called by the respective symbol. The *mName* member is a pointer to a null-terminated character string containing the name of the called symbol. The number of times that the symbol is referenced is given by *mReferenceCount*. The untransformed bounding box of the symbol is given by *mLeft*, *mBottom*, *mRight*, and *mTop*. Note that if symbol *mName* was opened by an invocation of the *CDOpen()* routine, the bounding box of the returned symbol descriptor would be identical to *mLeft*, *mBottom*, *mRight*, and *mTop*.

The purpose of the master-list is to simplify the procedure of reflecting any change to the bounding box of an instance in the bounding box of all its masters. For a brief example, consider symbols **X** and **Y** that contain an instance of symbol **Z**. If the bounding box of symbol **Z** is changed, a call to the *CDReflect()* procedure with the symbol descriptor of **Z** as an argument would cause the master-list of every resident symbol in the CD database, except symbol **Z**, to be searched for an occurrence of **Z**. When an instance of **Z** is found, the bounding box of the instance, as specified in the respective master-list structure member, is compared with the computed bounding box of symbol Z; if there is a difference, the master-list is modified and the bounding box of the master symbol is recomputed. At the completion of this procedure, the bounding boxes of symbols X and Y will have been recomputed to reflect the change in the bounding box of Z. In other words, the *CDReflect()* routine has the purpose of performing *bounding box propagation* which is further explained in Section 2.4.9 on CD integrity.

The symbol call descriptor also contains a pointer to a master-list descriptor. This is further explained in Section 2.4.4 that describes the CD object creation routines.

### 2.2.3. The Symbol Table Descriptor

The CD symbol table and symbol table descriptor are defined in the *cd.h* file as follows:

```
/*
 * Hash table of symbol descs keyed on symbol's name.
 */
struct bu {
     struct s *buSymbolDesc;
     struct bu *buPred;
     struct bu *buSucc;
     };
struct bu *CDSymbolTable[CDNUMLAYERS+1];
```

The procedure by which CD remembers a resident symbol is a hash table of symbol descriptors that is keyed on the name of the respective symbol. Each entry in the hash table named *CDSymbolTable* is a null-terminated, doubly linked-list of *bu* structures that contain a pointer to an open symbol descriptor. A procedure for computing the entry point into the hash table is shown below:

```
#include "cd.h"
/*
 * Function GetSymbol()
 * Find and return the symbol descriptor of symbol 'Name'.
 */
struct s *GetSymbol(Name)
        char *Name;             /* name of the symbol to be found */
        {
        int Key;
        char *cp
        struct bu *Bucket;

        cp = Name;
        while(*cp != NULL)
                Key += (int)( *cp++ );
        Bucket = CDSymbolTable[ Key % CDNUMLAYERS ];
        while(Bucket != NULL){
                if(strcmp(Bucket->buSymbolDesc->sName, Name) == 0)
                        return(Bucket->buSymbolDesc);
                Bucket = Bucket->buSucc;
                }
        }
```

### 2.2.4. The Object Descriptor

The object descriptor is defined in the *cd.h* file as follows:

```
/*
 * Object desc.
 */
struct o {
        int oLeft, oBottom, oRight, oTop;
        char oType;
        char oLayer;
        short oInfo;
        struct o *oPred, *oSucc;
        struct o *oRep;
        struct prpty *oPrptyList;
        };
```

One object descriptor is associated with each object in a symbol. This descriptor is allocated and initialized by an object-specific routine (e.g., *CDMakeBox()*, *CDMakeWire()*, *CDMakePolygon()*, etc. ). An object descriptor is released only by invocation of the *CDDeleteObjectDesc()* routine.

The *oType* member is a character that identifies the type of the object and can assume any of the following values defined in the *cd.h* file:

```
/*
 * Types of geometries
 */
#define    CDSYMBOLCALL       'c'
#define    CDPOLYGON          'p'
#define    CDROUNDFLASH       'r'
#define    CDLABEL            'l'
#define    CDWIRE             'w'
#define    CDBOX              'b'
```

The bounding box of the object is given by *oLeft*, *oBottom*, *oRight*, and *oTop*. The layer on which the object exists is specified by *oLayer*. When the object descriptor is allocated, the *oRep* pointer is cast as a pointer to an object-specific descriptor that extends the definition of the respective object; these object specific descriptors are defined in Section 2.4.4 concerns the object creation routines.

*oInfo* is the integer information field of the symbol and is initialized to zero, and *oPrptyList* is a pointer to the null-terminated object property list. The pointers *oPred* and *oSucc* are used to access the previous and successive objects in the doubly linked-list of objects that constitute a storage bin.

### 2.2.5. The Generator Descriptor

The generator descriptor is defined in the *cd.h* file as follows:

```
/*
 * Generator desc.
 */
struct g {
      int gLeft, gBottom, gRight, gTop;
      int gBeginX, gX, gEndX, gBeginY, gY, gEndY;
      char gLayer;
      struct o *gPointer;
      };
```

An object *generator* in the context of CD is a set of procedures for acquiring object descriptors for all objects on a specific layer and in a specific area of interest. Invocation of the *CDInitGen()* will return a pointer to an allocated and initialized generator descriptor that contains all the status information for the operation of the generator. More specifically, the generator descriptor stores information that includes the area of interest, the storage bins that are to be searched, and the present position in the storage bin currently being searched.

*gLeft*, *gBottom*, *gRight*, and *gTop* defines the area where the generated objects should reside, and *gLayer* is the layer on which the generated objects should exist with layer zero being the instance layer. The *gBeginX*, *gEndX* and *gBeginY*, *gEndY* members define the range of the storage bins to be searched, and *gX* and *gY* define the storage bin that is currently being searched. The *gPointer* member is a pointer to an object descriptor in the storage bins and points to the descriptor where the next search sequence will begin.

### 2.2.6. The Transform Descriptor

The CD package uses only rectilinear or "Manhattan" transformations. In CD, a transformation is characterized by a null-terminated linked-list of transformation descriptors. The transform descriptor is defined in the *cd.h* file as follows:

```
/*
 * Transform desc.
 * If MX, tType == CDMIRRORX.  If MY, tType == CDMIRRORX.
 * If R, tType == CDROTATE, tX == XDirection, tY == YDirection.
 * If T, tType == CDTRANSLATE, tX == TX, tY = TY;
 */
struct t {
    char tType;
    int tX, tY;
    struct t *tSucc;
    };
```

The transformation that is called *Mirroring in X* is an operation that effectively multiplies all x-coordinates by -1, or, in other words, mirrors in the direction of x. This transformation can be confusing because it is also identical to a reflection in the y-axis. Likewise, the transformation that is called *Mirroring in Y* is an operation that effectively multiplies all y-coordinates by -1, or, in other words, mirrors in the direction of y. This transformation is also identical to a reflection in the x-axis.

A rotation is always in the counter-clockwise direction and about the origin of the coordinate system. The angle of rotation is defined with two integers, namely *XDirection* and *YDirection*, and is the arctangent of the ratio *YDirection/XDirection*. If the angles of rotation is always an integer multiple of 90 degrees, either *XDirection* or *YDirection* will be zero, but never both.

The sequence of transformations is significant. A brief example of this fact follows: consider the point (0,0) translated in the positive x-direction by 100 units and then rotated 90 degrees. The resulting coordinate would be (0,100). Now consider the point (0,0) rotated by 90 degrees and then translated in the positive x-direction by 100 units, the exact opposite of the previous transformation order. The resulting point would be (100,0) and not (0,100).

The sequence of the transforms is identical to the succession of the transformation descriptors in the list. The character member *tType* specifies the type of transformation that is defined by the descriptor and can be assigned to the following values defined in the *cd.h* file:

```
/*
 * Types of transformations
 */
#define   CDMIRRORX       'x'   /* mirror in direction of x */
#define   CDMIRRORY       'y'   /* mirror in direction of y */
#define   CDROTATE        'r'   /* rotate by vector X, Y */
#define   CDTRANSLATE     't'   /* translate to X, Y */
```

If the transformation is a rotation or a translation, *tX* and *tY* define the angle or displacement, respectively. The *tSucc* member points to the next transformation descriptor in the linked-list and is null in the last transform definition in the list.

### 2.2.7. The Property List Descriptor

CD provides the capability of attaching property lists to symbols and objects; a CD property list is a null-terminated linked-list of property descriptors. The property list descriptor is defined in the *cd.h* file as follows:

```
/*
 * Property List desc.
 */
struct prpty {
      int prpty_Value;
      char *prpty_String;
      struct prpty *prpty_Succ;
      };
```

In CD, a property consists of an identifying integer and a null-terminated character string extension. The identifying integer *prpty_Value* defines the type of property, and the string extension *prpty_String* is a modifier. There is no standard set of properties and associated property values for CD or KIC; a user is free to set his own standard set of property values for his own specific use.

The *prpty_Succ* member points to the next property descriptor in the linked-list and is null in the last property in the list.

### 2.2.8. The CD Layer Descriptor

The layer descriptor is defined in the *cd.h* file as follows:

```
#define    CDNUMLAYERS       35
/*
 * CD Layer Table
 */
struct l {
      char lTechnology;
      char lMask[3];
      /* True if CDFrom should output layer */
      char lCDFrom;
      }
CDLayer[CDNUMLAYERS+1];
```

The CD layer descriptor is the building block of CD's layer table that is used to equate a layer's name to a positive integer. This integer is always used internally in CD to represent the respective layer.

A CIF layer name can be up to four characters long. The first character of the CIF layer name is stored in *lTechnology* and the remaining characters are stored in the *lMask* character buffer; the remaining characters in *lMask* will be blanks. A layer is recognized as undefined if the *lTechnology* character member of the layer table is a blank or space character. The *lCDFrom* boolean has the purpose of identifying the layer as visible or invisible when converting from KIC/CD format to another layout language, such as CIF or Calma STREAM.

The maximum layers known to CD, and therefore the size of the layer table, is defined by CDNUMLAYERS in the *cd.h* file. *CD* can be recompiled to provide more mask layers. Because object layer numbers are stored in character fields, the absolute maximum number of layers is 126. The size of the layer table is larger than CDNUMLAYERS because layer zero is reserved as the instance layer.

### 2.2.9. The Path Descriptor

A contour or trajectory of coordinates is represented in CD as a null-terminated linked-list of path descriptors. The path descriptor is defined in the *cd.h* file as follows:

```
/*
 * Linked path structure.
 */
struct p {
     int pX, pY;
     struct p *pSucc;
     };
```

The *pX* and *pY* integer members define one coordinate in the path. The sequence of the coordinates in the path is identical to the succession of the path descriptors in the linked-list. The *pSucc* member points to the next path descriptor in the linked-list and is null at the last coordinate in the list.

## 2.3. Storage Bins

All objects contained within a CD symbol are sorted by position and layer, and the sorted objects are stored in *bins*. A pointer to these storage bins is contained in the symbol descriptor, and each symbol has one complete set of storage bins. This section presents a detailed description of the storage algorithm.

The world coordinate system of CD ranges from CDBINMINX to CDBINMAXX in the x-direction, and from CDBINMINY to CDBINMAXY, and the world coordinate system is covered by a square array of CDNUMBINS by CDNUMBINS storage bins where these values are defined in the *cd.h* file as shown below. There is a special storage bin called the ***residual bin*** that only for convenience is contained in the storage bin array thereby adding another row and column to this array.

```
/*
 * These are the numbers that CD uses to determine which bin an object
 * resides in. They should reflect the average size of a layout being
 * edited by KIC. KIC will not fail if the numbers are too small.
 * Anything outside of this window is placed in the residual bin.
 * If these numbers become too large, CDIntersect() must use floating
 * point calculations.
 */
#define    CDBINMAXX           500000
#define    CDBINMAXY           500000
#define    CDBINMINX           (-CDBINMAXX)
#define    CDBINMINY           (-CDBINMAXY)

#define    CDNUMBINS           10
```

Each storage bin contains the beginning pointer to a linked-list of object descriptors, those objects that are contained in the particular bin. When an object is inserted into a bin as a result of a call to an object creation routines (e.g., *CDMakeBox()*, *CDMakeWire()*, etc.), it is inserted at the top of the linked-list. The bin into which an object is inserted depends on the bounding box of the object. If the bounding box intersects an area covered by more than one bin, the object is inserted into the residual bin. Otherwise, the object is inserted into the bin that contains the bounding box of the object.

If CD is compiled with a large number of bins, it will be able to rapidly traverse a symbol hierarchy or quickly access any geometry in the symbol. However, many storage bins also means that a significant amount of memory must be dedicated to the bin pointers, and that the number of objects in the residual bin to be disproportionately large when compared to the number of objects in the remaining bins; because the bins are smaller, the number of objects that intersect more than one bin increases and these objects are inserted into the residual bin.

The size of the residual bin can effect the speed at which the bins are searched for objects. When the storage bins are searched for an object in a specific area, it is of course necessary to examine the contents of each storage bin that intersects the particular area. However, the bounding box of an object that also intersects this area may also intersect several bins, in which case the object descriptor is stored in the residual bin. Therefore, the residual bin must always be searched, and consequently it is preferable to have the residual bin be as small as possible.

CD allocates the memory for the storage bin pointers on demand. This means that the memory requirements for the storage bins is a function of the number of layers being used in the symbol as well as the number of storage bins. When CD must allocate memory for the storage bin pointers, CDNUMBINS+1 by CDNUMBINS+1 bins are allocated, where CDNUMBINS is defined in the *cd.h* file. The bin that is pointed to by *sBin*[*Layer*][*0*][*0*] is the residual bin. The bins *sBin*[*Layer*][*X*][*0*] and *sBin*[*Layer*][*0*][*Y*], where *X* and *Y* are positive integers not greater than CDNUMBINS, are unused. The remaining bins represent the symbols storage bin structure.

The CD procedure that computes the range of storage bins that intersect a particular area is called *CDIntersect()*. The CD procedure that inserts an object descriptor into the storage bin of a symbol descriptor and allocates memory if the particular bin has not been allocated is called *CDInsertObjectDesc()*. These two CD routines are intended to be transparent to the CD user, and therefore will not be described further in this document.

To possibly increase the efficiency of this binning algorithm, the values of CDBINMAXX, CDBINMINX, CDBINMAXY, and CDBINMINY should be as small as possible while still reflecting a realistic working area for the CD application. CD will not fail if the real working area exceeds CD's range for the world coordinate system. Objects that exist outside the working world coordinate system of CD will be inserted into the outer-most bins, and an object generator will find such objects in those bins.

To minimize the number of objects in a given bin, the value of CDNUMBINS should be as large as is affordable with the resulting memory requirements. If the size of a pointer is four bytes, then the amount of memory required for only the pointers of a binning structure in an entire symbol hierarchy is given by the following equation:

4*(number of symbols)*(number of layers)*(CDNUMBINS+1)*(CDNUMBINS+1)

For an IC layout with 80 symbols, eight mask layers, and using only 10 bins, the memory requirements for the bin structure can exceed 300 kbytes. Also, as the numbers of bins increases, the size of the residual bin can be expected to increase. But by far the greatest disadvantage to the binning algorithm is that most bins remain empty and unused; a layout can typically exist entirely in the first Cartesian quadrant of the world coordinate system, in which case only one quarter of the bins would be used.

## 2.4. CD Procedures

### 2.4.1. CD Initialization

There are three procedures for initialization of the CD database.

```
void CDInit()

int CDPath(Path)
char *Path;

void CDSetLayer(Layer, Technology, Mask)
int Layer;
char Technology;
char Mask[3];
```

*CDInit()* must be invoked before any other CD procedures. This routine will clear the layer table, set the directory search path to be the present working directory, initialize the transformation stack, and allocate storage for diagnostics. *CDInit()* should be called only once by an application.

*CDPath()* sets the search rules for symbol-name resolution. The argument *Path* is a pointer to a null-terminated string containing a list of directory names to be searched separated by blanks. When a cell is opened via *CDOpen()* the list of directories is searched for the existence of the symbol file. The old search path is removed by a call to *CDPath()* and the default search path is the current directory. In the UNIX environment, *csh(1)* style names, as described in the BSD UNIX programmer's manual, will be understood.

*CDSetLayer()* Inserts the layer argument *Layer* into the CD layer table with the name *TechnologyMask*. If *Technology* and *Mask* contain only space or blank characters, the layer definition is removed from CD, but any object that exists on the particular layer is not deleted. The layer table is defined above with the CD layer descriptor.

### 2.4.2. Error Handling in CD

CD has a simple mechanism for handling errors. In the *cd.h* file, there are two external error variables that are allocated by *CDInit()*:

```
extern  int     CDStatusInt;        /* CD's diagnostic integer */
extern  char    *CDStatusString;    /* CD's status string */
```

If a CD routine encounters any difficulty, it will place an identifying diagnostic integer in *CDStatusInt* and a pointer to diagnostic character string in *CDStatusString*, and then return the with value of *False*. The possible fatal values for *CDStatusInt* are defined in the *cd.h* file as follows:

```
#define  CDPARSEFAILED       1    /* (FATAL) parse failed */
#define  CDNEWSYMBOL         3    /* symbol not in search path */
#define  CDMALLOCFAILED      11   /* (FATAL) out of memory */
#define  CDBADBOX            12   /* zero width or length box */
#define  CDXFORMSTACKFULL    13   /* transform stack overflow */
#define  CDBADPATH           14   /* bad directory search path */
```

Error handling in CD may be confusing at first because only those routines in which an error might occur will have a returned value. The routines in which no error is expected are assigned the type definition *void*.

### 2.4.3. CD Symbol Management

There are four CD procedures for creating, deleting, and maintaining CD symbols.

```
int CDOpen(SymbolName, SymbolDesc, Access)
char *SymbolName, Access;
struct s **SymbolDesc;

void CDSymbol(SymbolName, SymbolDesc)
char *SymbolName;
struct s **SymbolDesc;

int CDClose(SymbolDesc)
struct s *SymbolDesc;

int CDUpdate(SymbolDesc, SymbolName)
struct s *SymbolDesc;
char *SymbolName;
```

The procedure *CDOpen()* opens a symbol named *SymbolName*, allocates memory for and returns the pointer *SymbolDesc* to a symbol descriptor that represents the new symbol in the CD database.

The *Access* argument to *CDOpen()* is a character that determines the result after the current search path has been examined for the existence of a symbol named *SymbolName*. If the character *Access* equals the character 'w', then *CDOpen()* will create the cell in the database if it does not exist in the current search path. In other words, the symbol will be opened for writing. If *Access* equals the character 'r', then *CDOpen()* will create the cell in the database if and only if it exists in the current search path. In other words, the symbol is only read into memory. If the cell does not exist in the current search path, no symbol descriptor is inserted into the database, and *SymbolDesc* is assigned the value of NULL. Finally, if *Access* equals the character 'n', the symbol is opened regardless of whether any symbol named *SymbolName* exists in the current search path. If such a file exists in the search path, it is not read into memory. In other words, CD creates a new symbol.

*CDOpen()* will call the routine *PCIF()* to read the symbol into the database. The parsing procedure is described in greater detail in Chapter 3. A brief synopsis of *PCIF()* is as follows:

```
PCIF(SymbolName, StatusString, StatusInt)
char *SymbolName;
char **StatusInt;
int *StatusInt;
```

There are three requirements for the parser; first, the parser must locate and read the symbol *SymbolName*, and insert the symbol definition into the CD database by using the object creation routines described below in Section 2.4.4 (e.g., *CDMakeBox()*, *CDMakeWire()*, etc.). Second, the parser must provide a file called *parser.h* which contains the diagnostics described below. Finally, when the parse is completed, *PCIF()* must return a pointer to a null-terminated diagnostic string via *StatusString*, and *StatusInt* must be set to a value defined in the *parser.h* file as follows:

```
#define   PSUCCEEDED        1    /* successful return */
#define   PFAILED           2    /* parser failed */
#define   PNOTAPPLICABLE    3    /* parser failed, bad syntax */
```

*PCIF()* may return with the diagnostic string *StatusString* equal to NULL if and only if the parse succeeds. *CDOpen()* returns with the value *False* if the parse failed or if it was unable to allocate memory. When *CDOpen()* returns, *CDStatusInt*, as defined above, is set to a diagnostic value that is defined in the *cd.h* file as follows:

```
#define   CDPARSEFAILED     1    /* (FATAL) parse failed */
#define   CDOLDSYMBOL       2    /* symbol already in opened */
#define   CDNEWSYMBOL       3    /* symbol not in search path */
#define   CDSUCCEEDED       4    /* new symbol found in path */
```

Only CDPARSEFAILED is returned as a fatal error (i.e., *CDOpen()* returns with the value *False*, and *CDStatusInt* has been set to the value of CDPARSEFAILED); this simplifies the diagnostic test. However, if the *Access* argument is set to 'r' and the symbol is not found in the search path, *CDOpen()* returns with *CDStatusInt* set to the value of CDNEWSYMBOL. The application programmer should be aware of this behavior because it could be considered as an error if, for any reason, the symbol is expected to exist, such as when traversing a symbol hierarchy. When the returned value of *CDOpen()* identifies an error, *CDStatusString* will contain a diagnostic message.

*CDSymbol()* returns a pointer to symbol descriptor if the symbol *SymbolName* has been previously opened and exists in memory. If the symbol does not exist in memory, *SymbolDesc* is returned as a null pointer.

The *CDClose()* procedure will remove the symbol identified by *SymbolDesc* from memory and any associated instances and geometries. Also, all property lists associated with the symbol or any object in the symbol will be removed.

*CDUpdate()* will save any changes that have been made to the symbol referenced by *SymbolDesc*. The output will appear as CIF in a file in the current working directory identified by the null-terminated string *SymbolName*. If *SymbolName* is a null pointer, the name of the CIF file will be identical to the name of the symbol being updated.

### 2.4.4. CD Object Creation Routines

There are eight procedure for inserting an object into a CD symbol.

```
int CDMakeBox(SymbolDesc, Layer, Length, Width, X, Y, Pointer)
struct s *SymbolDesc;
struct o **Pointer;
int Layer;
int Length, Width;
int X, Y;

int CDMakeLabel(SymbolDesc, Layer, Label, X, Y, Pointer)
struct s *SymbolDesc;
struct o **Pointer;
int Layer;
char *Label;
int X, Y;

int CDMakePolygon(SymbolDesc, Layer, Path, Pointer)
struct s *SymbolDesc;
struct o **Pointer;
struct p *Path;
int Layer;

int CDMakeWire(SymbolDesc, Layer, Width, Path, Pointer)
struct s *SymbolDesc;
struct o **Pointer;
struct p *Path;
int Layer, Width;

int CDMakeRoundFlash(SymbolDesc, Layer, Width, X, Y, Pointer)
struct s *SymbolDesc;
struct o **Pointer;
int Layer;
int Width, X, Y;

int CDBeginMakeCall(SymbolDesc, SymbolName, NumX, DX, NumY, DY, Pointer)
struct s *SymbolDesc;
struct o **Pointer;
char *SymbolName;
int NumX, DX, NumY, DY;

int CDT(Pointer, Type, X, Y)
struct o *Pointer;
char Type;
int X, Y;

int CDEndMakeCall(SymbolDesc, Pointer)
struct s *SymbolDesc;
struct o *Pointer;
```

*CDMakeBox()* will create a box of length *Length* in the x direction and width *Width* in the y direction, centered at *X*,Y on the layer *Layer* in the symbol identified by the descriptor *SymbolDesc*. Zero width or length boxes are not allowed. *Pointer* contains a returned pointer to the object descriptor of the newly created box. Because the object descriptor contains sufficient information to characterize a rectangle, the *oRep* member of the object descriptor, as described previously in Section 2.2.4, is a null pointer if the object descriptor represents a box. There is no box descriptor in CD.

*CDMakeBox()* will return with the value *False* if it is unable to allocate storage. Otherwise, the value *True* is returned.

*CDMakeLabel()* will create a label on the layer *Layer* in the symbol identified by the descriptor *SymbolDesc*. The lower, left corner of the label will be referenced to the coordinate *X*, Y, and *Label* is a pointer to a null-terminated string containing the label. *Pointer* contains a returned pointer to the object descriptor of the newly created label. For all labels, the *oRep* member of the object descriptor, as described in Section 2.2.4, is cast as a pointer to a label descriptor that is defined in the *cd.h* file as follows:

```
/*
 * Label desc.
 */
struct la {
     char *laLabel;            /* text body */
     int laX, laY;             /* lower, left corner */
     };
```

*CDMakeLabel()* will return with the value *False* if it is unable to allocate storage. Otherwise, the value *True* is returned.

*CDMakePolygon()* will create a polygon with a linked-list coordinate path *Path* on the layer *Layer* in the symbol identified by the descriptor *SymbolDesc*. *Path* is a pointer to a linked-list of coordinates as described previously in the section that defines the path descriptor. *Pointer* contains a returned pointer to the object descriptor of the newly created polygon. For all polygons, the *oRep* member of the object descriptor, as described in Section 2.2.4, is cast as a pointer to a polygon descriptor that is defined in the *cd.h* file as follows:

```
/*
 * Polygon desc.
 */
struct po {
     struct p *poPath;          /* the polygon contour */
     };
```

*CDMakePolygon()* will return with the value *False* if it is unable to allocate storage. Otherwise, the value *True* is returned.

*CDMakeWire()* will create a wire of width *Width* with a coordinate path *Path* on the layer *Layer* in the symbol referenced by the descriptor *SymbolDesc*. *Path* is a pointer to a linked-list of coordinates as described above in Section 2.2.9 that concerns the CD path descriptor. *Pointer* contains a returned pointer to the object descriptor of the newly created wire. For all wires, the *oRep* member of the object descriptor, as described in Section 2.2.4, is cast as a pointer to a wire descriptor that is defined in the *cd.h* file as follows:

```
/*
 * Wire desc.
 */
struct w {
     int wWidth;                /* the wire width */
     struct p *wPath;           /* the wire contour */
     };
```

*CDMakeWire()* will return with the value *False* if it is unable to allocate storage. Otherwise, the value *True* is returned.

*CDMakeRoundFlash()* will create a round flash of diameter *Width* centered at *X*, Y on the layer *Layer* in the symbol identified by the descriptor *SymbolDesc*. Zero diameter round flashes are not allowed. *Pointer* contains a returned pointer to the object descriptor of the newly created round flash. For all round flashes, the *oRep* member of the object descriptor, as described in Section 2.2.4, is cast as a pointer to a round flash descriptor that is defined in the *cd.h* file as follows:

```
/*
 * Round flash desc.
 */
struct r {
     int rWidth;              /* the flash diameter */
     int rX, rY;              /* the center of the flash */
     };
```

The KIC program does not use round flashes and the round flash routines are only presented in this section for completeness. All round geometries are represented in KIC as polygons.

*CDMakeRoundFlash()* will return with the value *False* if it is unable to allocate storage. Otherwise, the value *True* is returned.

*CDBeginMakeCall()* allocates memory and initializes the object descriptor that will represent the newly created instance of the symbol *SymbolName*. *NumX* is the number of instances in the untransformed x direction and *NumY* is the number of instances in the untransformed y direction. *DX* and *DY* are the distances between the left and right edges and the top and bottom edges respectively of two adjacent cells in the instance array. *Pointer* returns a pointer to the new object descriptor. For all symbol calls, the *oRep* member of the object descriptor, as described in Section 2.2.4, is cast as a pointer to a call descriptor that is defined in the *cd.h* file as follows:

```
/*
 * Call desc.
 */
struct c {
     /* not used */
     int cNum;
     /* Pointer to transformation descriptor. */
     struct t *cT;
     /* Pointer to master-list descriptor. */
     struct m *cMaster;
     /* Array parameters. */
     int cNumX, cNumY, cDX, cDY;
     };
```

The *cT* member of the call descriptor is a pointer to the transformation list that will be defined by subsequent calls to the *CDT()* routine. The *cMaster* member is a pointer to the master-list descriptor in the master-list of the symbol that instances the cell represented by the call descriptor. Because the master-list descriptor contains the instance name, it is not necessary to contain a name buffer in the call descriptor. The master-list is described in Section 2.2.2.

If *SymbolName* is not in the current search path, or *CDBeginMakeCall* cannot allocate storage, *CDBeginMakeCall* returns with the value *False* and *CDStatusInt* will be set to a diagnostic integer defined in the *cd.h* file as follows:

```
#define   CDPARSEFAILED      1    /* (FATAL) parse failed */
#define   CDNEWSYMBOL        3    /* symbol not in search path */
#define   CDMALLOCFAILED     11   /* (FATAL) out of memory */
```

After invoking *BeginMakeCall()*, the procedure *CDT()* is invoked for each transformation in the call. The argument *Pointer* is a pointer to the object descriptor that was returned by the *CDBeginMakeCall()* procedure. The character *Type* identifies the transformation to be added to the call, and the valid arguments for *Type* are defined in the *cd.h* file as follows:

```
#define   CDMIRRORX          'x'  /* mirror in direction of x */
#define   CDMIRRORY          'y'  /* mirror in direction of y */
#define   CDROTATE           'r'  /* rotate by vector X, Y */
#define   CDTRANSLATE        't'  /* translate to X, Y */
```

The arguments *X* and *Y* of *CDT()* are used to qualify a rotation or translation. Only rectilinear rotations are valid. For a rotation of 90 degrees, *X* has the value of 1, and *Y* has the value of 0. For a rotation of 180 degrees, *X* has the value of -1, and *Y* has the value of 0. For a rotation of 270 degrees, *X* has the value of 0, and *Y* has the value of -1. For a translation, the integers *X* and *Y* are used to specify the x and y displacements, respectively.

Remember that *transformation order is significant*. Each invocation of *CDT()* adds another transformation descriptor to the transformation list that is referenced by *cT* in the respective call descriptor, and thus the invocations of

*CDT()* must be in the identical order of the instance transformation.

Finally, *CDEndMakeCall()* is invoked to insert the call into the master symbol identified by *SymbolDesc*. The argument *Pointer* was of course returned by a previous call to the *CDBeginMakeCall()* procedure. *CDEndMakeCall()* will return with the value *False* if it is unable to allocate storage. Otherwise, the value *True* is returned.

### 2.4.5. CD Object Generator

An object *generator* in the context of CD is a set of procedures for acquiring object descriptors for all objects on a specific layer and in a specific area of interest. A generator will contain the following three CD procedures:

```
int CDInitGen(SymbolDesc, Layer, Left, Bottom, Right, Top, GenDesc)
struct s *SymbolDesc;
struct g **GenDesc;
int Layer;
int Left, Bottom;
int Right, Top;

void CDGen(SymbolDesc, GenDesc, Pointer)
struct s *SymbolDesc;
struct g *GenDesc;
struct o **Pointer;

void CDType(Pointer, Type)
struct o *Pointer;
char *Type;
```

*CDInitGen()* returns a pointer to a generator storage descriptor *GenDesc* that is allocated automatically and described in Section 2.2.5. Further invocations of *CDGen()* will return each object in the symbol identified by *SymbolDesc* on the layer *Layer* whose bounding box intersects the area given by *Left*, *Bottom*, *Right*, and *Top*. If *Layer* equals zero, the invocations of *CDGen()* will return instances only (i.e., layer zero is the instance layer). *CDInitGen()* will return with the value of *False* if it is unable to allocate the generator storage descriptor. Otherwise, the value of *True* is returned.

*CDGen()* returns a pointer to an object descriptor that identifies an object within the area of interest as defined by the previous call to *CDInitGen()* which returned the pointer to the generator descriptor *GenDesc*. If *CDGen()* returns with *Pointer* set to NULL, then the last object has been returned and *GenDesc* storage has been released.

*CDType()* returns the type of an object pointed to by *Pointer*. The argument *Type* is a pointer to a character that is set by *CDOpen()* to a value defined as follows in the *cd.h* file:

```
/*
 * Types of geometries
 */
#define    CDSYMBOLCALL        'c'
#define    CDPOLYGON           'p'
#define    CDROUNDFLASH        'r'
#define    CDLABEL             'l'
#define    CDWIRE              'w'
#define    CDBOX               'b'
```

This information routine is typically used in a generator loop to dispatch a type-specific procedure for accessing the object. An example of a generator loop for traversing a symbol hierarchy is provided in Section 2.6.

### 2.4.6. Accessing Objects in CD

All information that is necessary to characterize an object is stored in that object's respective object descriptor and another representative descriptor that is referenced in the object descriptor. Information for a specific object is therefore easily obtained from the respective object descriptor. The following eight CD procedures are used for accessing object-specific information:

```
void CDBox(Pointer, Layer, Length, Width, X, Y)
struct o *Pointer;
int *Layer, *Length, *Width, *X, *Y

void CDLabel(Pointer, Layer, Label, X, Y)
struct o *Pointer;
char **Label;
int *Layer;
int *X, *Y;
```

```
void CDPolygon(Pointer, Layer, Path)
struct o *Pointer;
int *Layer;
struct p **Path;

void CDWire(Pointer, Layer, Width, Path)
struct o *Pointer;
struct p **Path;
int *Layer, *Width;

void CDRoundFlash(Pointer, Layer, Width, X, Y)
struct o *Pointer;
int *Layer;
int *Width, *X, *Y;

void CDCall(Pointer, SymbolName, NumX, DX, NumY, DY)
struct o *Pointer;
char **SymbolName;
int *NumX, *DX, *NumY, *DY;

void CDInitTGen(Pointer, TGen)
struct o *Pointer;
struct t **TGen;

void CDTGen(TGen, Type, X, Y)
struct t **TGen;
char *Type;
int *X, *Y;
```

*CDBox()* will return the length *Length* in the x direction and the width *Width* in the y direction of a box identified by the pointer *Pointer* to an object descriptor. The box is centered at the coordinate *X*, *Y* and on the layer *Layer*.

*CDLabel()* returns the pointer to a null-terminated label *Label* that has lower, left justification to the coordinate *X*, *Y* and whose object descriptor is pointed to by *Pointer*. The label is on the layer *Layer*.

*CDPolygon()* will return a pointer to the linked-list coordinate path *Path* of a polygon identified by the pointer *Pointer* to an object descriptor. The polygon is on the layer *Layer*. The linked-list coordinate path is defined in Section 2.2.9 that describes the path descriptor.

*CDWire()* will return a pointer to the linked-list coordinate path *Path* of a wire with width *Width* that is identified by the pointer *Pointer* to an object descriptor. The wire is on the layer *Layer*.

*CDRoundFlash()* will return the diameter *Width* of a round flash identified by the pointer *Pointer* to an object descriptor. The round flash is centered at the coordinate *X, Y* and on the layer *Layer*. As explained earlier, KIC does not use CD round flash routines.

*CDCall()* returns the a character pointer to the name *SymbolName* of an instance referenced by the object descriptor *Pointer*. Also returned is *NumX* which is the number of instance in the untransformed x direction and *NumY* which is the number of instances in the untransformed y direction. *DX* and *DY* are the distances between the left and right edges and the top and bottom edges respectively of two adjacent cells in the instance array.

The *CDInitTGen()* routine initializes the transformation generator loop to access the transformations of the instance referenced by *Pointer*. *TGen* is a returned pointer to a transform generator descriptor. Subsequent invocations of *CDTGen()* will return the individual components of the instance transformations.

*CDTGen()* returns a character that identifies one component of the transformation of the instance for which *TGen* was returned by the *CDInitTGen()* routine as the transform generator descriptor. The argument *Type* is a pointer to a character that is set by *CDTGen()* to one of the values that are defined as follows in the *cd.h* file:

```
#define   CDMIRRORX       'x'   /* mirror in direction of x */
#define   CDMIRRORY       'y'   /* mirror in direction of y */
#define   CDROTATE        'r'   /* rotate by vector X, Y */
#define   CDTRANSLATE     't'   /* translate to X, Y */
```

The order in which the transformation components are returned by *CDTgen()* is identical to the order in which they were defined by calls to *CDT()*. The integer pointers *X, Y* are used to specify a rotation or translation. For a rotation of 90 degrees, *X* has the returned value of 0, and *Y* has the returned value of 1. For a rotation of 180 degrees, *X* has the returned value of -1, and *Y* has the returned value of 0. For a rotation of 270 degrees, *X* has the returned value of 0, and *Y* has the returned value of -1. For a translation, the returned values of *X* and *Y* specify the x and y displacements, respectively. When *CDTGen()* returns with *TGen* set to NULL, then the last transformation has been returned and *TGen* storage has been freed.

### 2.4.7. Object Deletion in CD

There is one procedure for removing objects from their respective symbol:

```
void CDDelete(SymbolDesc, Pointer)
struct s *SymbolDesc;
struct o *Pointer;
```

*CDDelete()* will remove the object pointed to by *Pointer* from the symbol referenced by *SymbolDesc*. Because the object descriptor is doubly linked-list (i.e., it contains a pointer to the successor and predecessor in the list), the task of deleting the object from the list can be completed without searching the storage bins for the particular object that is to be removed. Also, the memory of any property list that is associated with the deleted object is released.

### 2.4.8. CD Information Routines

There are six procedure for accessing or setting a variety of information fields in object or symbol descriptors:

```
int CDBB(SymbolDesc, Pointer, Left, Bottom, Right, Top)
struct s *SymbolDesc;
struct o *Pointer;
int *Left, *Bottom, *Right, *Top;

void CDInfo(SymbolDesc, Pointer, Info)
struct s *SymbolDesc;
struct o *Pointer;
int *Info;

void CDSetInfo(SymbolDesc, Pointer, Info)
struct s *SymbolDesc;
struct o *Pointer;
int Info;
```

```
void CDProperty(SymbolDesc, Pointer, Property)
struct s *SymbolDesc;
struct o *Pointer;
struct prpty **Property;

int CDAddProperty(SymbolDesc, Pointer, Value, String)
struct s *SymbolDesc;
struct o *Pointer;
char *String;
int Value;

int CDRemoveProperty(SymbolDesc, Pointer, Value)
struct s *SymbolDesc;
struct o *Pointer;
int Value;
```

*CDBB()* returns the bounding box of an object pointed to by *Pointer* in the symbol identified by *SymbolDesc*. If *Pointer* is a null pointer, then *CDBB()* returns the bounding box of the entire symbol. The *CDBB()* procedure may have to use temporary storage during the computation of a symbols bounding box. If it can not allocate the required memory, *CDBB()* returns with the value of *False*. Otherwise, the value of *True* is returned.

*CDInfo()* returns the value *Info* of the integer information field of an object referenced by *Pointer*. If *Pointer* is a null pointer, then the value of the information field of the symbol referenced by *SymbolDesc* is returned.

*CDSetInfo()* sets the integer information field the object pointed to by *Pointer*. If *Pointer* is a null pointer, then the information field of the symbol referenced by *SymbolDesc* is set. The information field is used extensively by KIC to mark the object or temporarily storing information while the object is being modified.

*CDProperty()* returns the pointer *Property* that references the linked-list of properties associated with the object referenced by *Pointer*. If *Pointer* is a null pointer, then the property list of the symbol referenced by *SymbolDesc* is returned. The pointer *Property* is returned as a null pointer if there is no property list associated with the particular object.

The property list structure is defined in Section 2.2.7. A property consists of an identifying integer and a null terminated character string extension. There is no standard set of properties or property values, and CD never tries to interpret the entries in a property list. Property lists are available for the user's own specific use.

*CDAddProperty()* inserts property information into the property list of the object referenced by *Pointer*. If *Pointer* is a null pointer, then the property is added to the property list of the symbol referenced by *SymbolDesc*. The property information consists of an identifying integer *Value* and a null-terminated character string extension that is pointed to by *String*. If *CDAddProperty()* is unable to allocate memory, the value *False* is returned. Otherwise, the value of *True* is returned.

*CDRemoveProperty()* deletes property information from the property list of the object referenced by *Pointer*. If *Pointer* is a null pointer, then the property is removed from the property list of the symbol referenced by *SymbolDesc*. Every property with the value of *Value* is removed. If *CDRemoveProperty()* has trouble allocating or releasing memory, the value *False* is returned. Otherwise, the value of *True* is returned.

### 2.4.9. CD Integrity

After a symbol has been created or modified, it is eventually necessary to reflect the changes throughout the CD database. The following procedure is provided for this purpose:

```
int CDReflect(SymbolDesc)
struct s *SymbolDesc;
```

*CDReflect()* must be invoked at certain times by the CD application if the symbol that is referenced by *SymbolDesc* is modified. If the changes to the symbol has changed its bounding box, a call to *CDReflect()* will update the bounding box information in every other symbol in the CD database that references it either directly or indirectly. This procedure is called *bounding box propagation*. Only the changes to the symbol referenced by *SymbolDesc* are propagated through the database.

The correct use of *CDReflect()* is important and well worth an example. Consider two symbols *X* and *Y*, where symbol *Y* is called in *X*. The bounding box of *Y* is changed by some means. If *CDReflect()* is invoked with the symbol descriptor of *X* as an argument, the changes to symbol *Y* will not be reflected in the bounding box of symbol *X*. If however *CDReflect()* is invoked with the symbol descriptor of *Y* as an argument, all changes to the symbol *Y* will then be reflected in the bounding box of *X* and all other symbols that call *Y*.

The value of *False* is returned if *CDReflect()* is unable to allocate new memory. Otherwise, the value of *True* is returned.

## 2.5. Two Dimensional Transformation Package

The following routines provide two dimensional, rectilinear transformations using integer arithmetic. The package of routines includes such capabilities as translation, mirroring, rotation, a transformation stack, and inverse transformations. Transformations are modeled by three by three integer matrices. A further explanation of these procedures and how a transformation may be modeled by an integer array is provided in [6] and [7].

### 2.5.1. Initialization

The following procedure initializes the transform package:

```
void TInit()
```

*TInit()* initializes the transform package and returns with the current transform equal to the identity transform. Unlike the *CDInit()* procedure, *TInit()* may be invoked more than once in an application program where the only effect will be that all contexts in the transformation stack are lost. This initialization routine is invoked by *CDInit()*, by KIC routines that capable of detecting a recursive symbol hierarchy, and by interrupt handling routines.

### 2.5.2. The Current Transformation

The following eight procedures are used to access or modify the current transformation:

```
void TIdentity()

void TTranslate(X, Y)
int X, Y;

void TMY()

void TMX()

void TRotate(XDirection, YDirection)
int XDirection, YDirection;

void TPoint(X, Y)
int *X, *Y;
```

*TIdentity()* sets the current transformation to the identity matrix. The previous current transformation is destroyed.

*TTranslate()* postmultiplies the current transformation matrix by a transformation that translates by a displacement of *X*, *Y*.

*TMY()* postmultiplies the current transformation matrix by a transformation that mirrors the y-coordinates (i.e., mirrors in the direction of the y axis).

*TMX()* postmultiplies the current transformation matrix by a transformation that mirrors the x-coordinates (i.e., mirrors in the direction of the x axis).

*TRotate()* postmultiplies the current transformation matrix by a transformation that rotates counter-clockwise by an angle that is expressed as a CIF-style direction vector. Only 0, 90, 180, 270 degree rotations are allowed.

The *TPoint()* procedure transforms the point *X*, *Y* by multiplying it by the current transformation matrix.

### 2.5.3. The Transformation Stack

The transformation stack structure is defined in the *xforms.h* file as follows:

```
#define    XFORMSTACKSIZE         100
/*
 * Transformation stack structure.
 */
struct tt {
     int ttStack[XFORMSTACKSIZE][3][3];
     int ttSP;
     int ttCurrent[3][3];
     int ttInverseCurrent[3][3];
     };
```

Transformations are saved in their respective three by three array representations in the *ttStack* stack structure member. The *ttSP* integer member points to the current transformation on the stack, and the current transformation is also contained in the *ttCurrent* array member. When the user invokes the *TInverse()* procedure to compute the inverse of the current transform, the resulting transformation matrix is saved in the *ttInverseCurrent* structure member.

The following procedures are used for maintaining the transformation stack:

```
int TEmpty()
int TFull()
void TPush()
void TPop()
void TCurrent(TF)
int *TF;
```

*TEmpty()* returns the value of *True* if the transformation stack is empty. If the transformation stack is completely filled, the value of *False* is returned.

*TFull()* returns the value of *True* if the transformation stack is full. Otherwise, the value of *False* is returned.

*TPush()* pushes the current transform onto the transformation stack. The value of the current transform is not changed, and the transformation stack is not checked for an overflow condition.

*TPop()* pops the current transform from the transformation stack. The value of the current transform becomes the transform that was most recently pushed onto the stack, and the transformation stack is not checked for an underflow condition.

*TCurrent()* places the current transform matrix in a nine integer array that is passed from the calling program. The first row of the transformation matrix appears as the first three integers in the argument, the second row of the transformation matrix appears as the next three integers, etc. After several transformations have been defined by *TTranslate()*, *TRotate()*, *TMX()*, and *TMY()*, it is possible to determine the minimum resultant or equivalent transformation through the examination of the elements of the current transformation matrix as described in the following table:

| TF[0] | TF[3] | TF[1] | TF[4] | Transformation |
|---|---|---|---|---|
| 1 | 0 | 0 | 1 | Translate only. |
| 0 | -1 | 1 | 0 | Rotate 90 deg., translate. |
| 0 | 1 | -1 | 0 | Rotate 180 deg., translate. |
| -1 | 0 | 0 | -1 | Rotate 270 deg., translate. |
| -1 | 0 | 0 | 1 | Mirror in X, translate. |
| 1 | 0 | 0 | -1 | Mirror in Y, translate. |
| 0 | -1 | -1 | 0 | Mirror in X, rotate 90 deg., translate. |
| 0 | 1 | 1 | 0 | Mirror in X, rotate 270 deg., translate. |

For all cases, the X, Y translation vector is given by TF[6], TF[7].

### 2.5.4. The Instance Transformation

The following procedure is provided specifically for maintaining instance transformations:

```
void TPremultiply()
```

As an application program traverses a symbol hierarchy, it will maintain the current instance transformation by computing the product of the symbols current transform and the transformation of its master. *TPremultiply()* forms the instance transform by premultiplying the current transform with the transform that was last pushed onto the transformation stack and placing the product in the current transformation matrix. Thus, the procedure for transforming the coordinates of an instance is demonstrated below:

```
/* push master cell transform onto stack */
TPush();

/* set current transform to identity */
TIdentity();

/* Invoke TMX, TTranslate, etc. to build instance transform */
TMX();
TMY();
TTranslate(Dx, Dy);

/* Form the instance transform */
TPremultiply();

/* Invoke TPoint to transform instance points */
TPoint(&X, &Y);

/* return to master transform */
TPop();
```

### 2.5.5. Inverse of the Current Transformation

The following procedures are used to obtain an inverse transformation:

```
void TInverse()

void TInversePoint(X, Y)
int *X, *Y;
```

*TInverse()* computes the inverse of the current transformation. Computation of the inverse transformation does not affect the current transformation. The *TInversePoint()* routine transforms the point *X*, *Y* by multiplying it by the inverse transform matrix. The *TInversePoint()* routine should not be invoked before the *TInverse()* procedure has computed the inverse of the current transformation.

## 2.6. Traversing a Symbol Hierarchy with CD

The following routine is intended to display a symbol on a graphics terminal. The reader will note the use of the two dimensional transformation package described above as well as routines such as *DisplayBox()* that display geometries on a CRT. It is assumed that the geometry display routines will accomplish whatever window-viewport clipping is necessary. This example is similar to the KIC display routine.

```
#include "cd.h"
main(){
        .
        .
        .
    /* initialize transformation stack */
    TInit();
        .
        .
        .
    /* display SymbolName in the area Left, Bottom - Right, Top */
    Display(SymbolName, Left, Bottom, Right, Top)
        .
        .
        .
    }
Display(SymbolName, Left, Bottom, Right, Top)
    char *SymbolName;
    int Left, Bottom, Right, Top;
    {
    struct s *SymbolDesc;
    struct g *GenDesc;
    struct o *Pointer;
    struct t *TGen;
    char *InstanceName;
    int NumX, NumY, DX, DY;
    int X, Y, Layer;
    char Type;

    /* open symbol named SymbolName (here we assume it exists) */
    CDOpen(SymbolName, &SymbolDesc, 'r');

    /* initialize generator to return instances in SymbolName */
    CDInitGen(SymbolDesc, 0, Left, Bottom, Right, Top, &GenDesc);

    loop {
        /* Invoke CDGen to access a pointer to an instance array */
        CDGen(SymbolDesc, GenDesc, &Pointer);
```

```
          /* Have all instances been traversed? */
          if(Pointer == NULL)
                break;

          /* push current transform of master onto stack */
          TPush();

          /* set current transform to identity */
          TIdentity();

          /* Access instance information */
          CDCall(Pointer, &InstanceName, &NumX, &DX, &NumY, &DY);

          /* Initialize generator to return transform of InstanceName */
          CDInitTGen(Pointer, &TGen);

          /* place instance transformation in current transformation */
          loop {
                CDTGen(&TGen, &Type, &X, &Y);
                if(TGen == NULL)
                      break;
                else if(Type == CDTRANSLATE)
                      TTranslate(X, Y);
                else if(Type == CDMIRRORX)
                      TMX();
                else if(Type == CDMIRRORY)
                      TMY();
                else if(Type == CDROTATE)
                      TRotate(X, Y);
                }

          /* Combine transform of InstanceName with it's master */
          TPremultiply();

          /* recursively call display to traverse and display instance */
          Display(InstanceName, Left, Bottom, Right, Top);

          /* pop master transform from stack */
          TPop();
          }
    /* now traverse the geometries */
    for(Layer = 1; Layer <= CDNUMLAYERS; ++Layer) {

          /* Set the current color to be a color associated with Layer. */
          SetColor(Layer);

          /* initialize generator for layer Layer */
          CDInitGen(SymbolDesc, Layer, Left, Bottom, Right, Top, &GenDesc);

          loop {
                /* Invoke CDGen to access pointer to an object */
                CDGen(SymbolDesc, GenDesc, &Pointer);

                /* Last object? */
                if(Pointer == NULL)
                      break;

                /* Access the type of the geometry as Type */
                CDType(Pointer, &Type);
```

```
        /* Dispatch according to Type to specific procedure */
        if(Type == CDBox) {
            /* Access the box */
            CDBox(SymbolDesc, Pointer, ...);
            /* Transform the box's center. */
            TPoint(&X, &Y);
            /* Display the box. */
            DisplayBox(X, Y, Length, Width);
            }
        else if(Type == CDWire){
            /* Access the wire */
            CDWire(SymbolDesc, Pointer, ...);
               .
               .
               .
            }
          .
          .
          etc.
          .
          .
        }
    }
}
```

## 2.7. Translation Routines

The layout description language of CD is CIF. The following routines are used by CD to generate or interpret CIF:

```
int CDTo(CIFFile,Root,A,B,Program)
char *CIFFile,*Root;
int A,B;
char Program;

int CDFrom(Root,CIFFile,A,B,Layers,NumLayers,Program)
char *Root,*CIFFile,Program;
int *Layer,NumLayers;
int A,B;

int CDParseCIF(Root,CIFFile,Program)
char *Root,*CIFFile,Program;

int CDGenCIF(FileDesc,SymbolDesc,SymbolNum,A,B,Program)
FILE *FileDesc;
struct s *SymbolDesc;
int *SymbolNum,A,B;
char Program;
```

*CDTo()* translates from a CIF file named *CIFFile* into symbol files, each having a file name identical to the symbol that it contains. CIF commands that are not between a *DS* and a matching *DF* are stored in the file specified by *Root*. All objects are scaled by the ratio *A/B* microns per lambda.

*CDTo()* will call the routine *PCIF()* to read the input CIF file. The requirements for this parser can be found in Section 2.4.3 describing the *CDOpen()* routine and in the section that describes the fast CIF parser.

Because there are different styles of CIF that embed symbol names differently, the character *Program* will tell *CDTo()* and the parser *PCIF()* which style of CIF to expect. Before calling the parser, *CDTo()* sets the *dProgram* character in the CD parameters structure described in Section 2.8 to the character *Program*. By accessing this structure member, the parser determines the origin of the CIF. The following values for *Program* are valid for the fast CIF parser used by CD:

| **Program** | **CIF format** |
|---|---|
| 'a' | Stanford CIF, |
| 'b' | NCA CIF, |
| 'e' | Berkeley's KIC with property extensions, |
| 'h' | HP's IGS, |
| 'i' | Xerox's Icarus, |
| 'k' | Berkeley's KIC, |
| 'm' | mextra-style CIF, |
| 's' | Xerox's Sif, |
| 'n' | none of the above. |

If the *CDTo()* routine encounters any difficulty in the CIF conversion, the value of *False* is retuned, *CDStatusInt* is set to value of *CDPARSEFAILED*, and a diagnostic message is placed in *CDStatusString*.

The *CDFrom()* routine translates a symbol hierarchy rooted at the symbol named *Root* into a CIF file named *CIFFile*. The style of CIF output is identified by the character *Program*. The valid arguments for *Program* are the same as for the *CDTo()* procedure. All objects are scaled by the ratio *A/B* microns per lambda during the conversion. It is assumed that all instances in the symbol hierarchy exist in the current search path.

The *Layers* argument is a pointer to an array of *NumLayers* integers that are used to mask certain layers in CD layer table. If *Layers*[*N*] is zero, where *N* is a non-negative integer less than *NumLayers*, then any object that is on layer number *N* in the CD layer table will not appear in the CIF output file.

If *CDFrom()* encounters any difficulty in the conversion, the value of *False* is retuned and *CDStatusInt* is set to one of the following values defined in the *cd.h* file:

```
#define   CDPARSEFAILED      1    /* (FATAL) parse failed */
#define   CDNEWSYMBOL        3    /* symbol not in search path */
#define   CDMALLOCFAILED    11    /* (FATAL) out of memory */
```

If no difficulty is encountered, *CDFrom()* returns with the value of *True*.

The *CDParseCIF()* procedure will create a CD database rooted at a symbol named *Root* from a CIF file *CIFFile* rather than building the database from a hierarchy of symbol files. The style of CIF input is identified by the character *Program*. The valid arguments for *Program* are the same as for the *CDTo()* routine.

When *CDParseCIF()* encounters a reference to a layer that was not previously defined in the CD layer table by a call to *CDSetLayer()*, the new layer is added to the layer table. This differs from the *CDFrom()* and *CDOpen()* routines that return CDPARSEFAILED whenever they encounter an undefined layer. A layer is considered undefined if the *lTechnology* field in the CD layer table is a blank character. See Section 2.2.8 that describes the CD layer descriptor.

If *CDParseCIF()* encounters any difficulty in the conversion, the value of *False* is retuned and *CDStatusInt* is set to one of the following values defined in the *cd.h* file:

```
#define   CDPARSEFAILED      1    /* (FATAL) parse failed */
#define   CDNEWSYMBOL        3    /* symbol not in search path */
#define   CDMALLOCFAILED     11   /* (FATAL) out of memory */
```

If no difficulty is encountered, *CDParseCIF* returns with the value of *True*.

*CDGenCIF()* is used by the *CDTo()* routine to generate a CIF file identified by the stream *FileDesc* of the CD symbol referenced by *SymbolDesc*. The integer *SymbolNum* contains the number of the first symbol created in the CIF file; the value will be incremented by one for succeeding symbol definitions. The style of CIF output is identified by the character *Program*, and the valid arguments for *Program* are the same as for the *CDTo()* procedure. All objects are scaled by the ratio *A/B* microns per lambda during the conversion. If *CDFrom()* encounters any difficulty in acquiring or allocating memory, the value of *False* is retuned and *CDStatusInt* is set to the value of CDMALLOCFAILED as defined in the *cd.h* file. If no difficulty is encountered, *CDGenCIF()* returns with the value of *True*.

### 2.8. The CD Parameters Descriptor

Several parameters are required by CD to control actions. The CD parameters struct is defined in the *cd.h* file as follows:

```
struct d {
        /*
         * DCONTROLCDOPEN denotes CD is in CDOpen
         * DCONTROLPCIF denotes CD is in parsing CIF in CDParseCIF
         * DCONTROLCDTO denotes CD is in CDTo
         * DCONTROLVANILLA denotes CD is in none of the above
         */
        char dControl;

        /* Current parameters for symbol being parsed in CDOpen. */
        int  dNumX,dDX,dNumY,dDY;

        /* Scale factors for CDTo and CDFrom. */
        int dA,dB;

        /* Symbol scale factors. */
        int dDSA,dDSB;

        struct o *dPointer;
        struct s *dSymbolDesc;
        struct s *dRootCellDesc;

        /* UNIX file names are limited to 14 characters */
        char dSymbolName[FILENAMESIZE];
        FILE *dSymbolFileDesc;

        /*
         * Fields used in CDTo follow.
         */

        /* True if parsing root symbol. */
        int dRoot;

        /* Root's file desc. */
        FILE *dRootFileDesc;

        /* Current property list for symbol being parsed */
        struct prpty *dPrptyList;

        /*
         *  dProgram  ==  'a' if Stanford CIF.
         *  dProgram  ==  'b' if NCA CIF.
         *  dProgram  ==  'e' if Berkeley CIF with property extensions.
         *  dProgram  ==  'h' if HP's IGS.
         *  dProgram  ==  'i' if Xerox's Icarus CIF.
         *  dProgram  ==  'k' if Berkeley CIF.
         *  dProgram  ==  'm' if mextra-style CIF.
         *  dProgram  ==  's' if Sif gened it.
         *  dProgram  ==  'n' if none of the above.
         */
        char dProgram;
        /*
         * Symbol name table for CIF file being parsed.
         * UNIX file names are 14 characters, VMS names are smaller.
         */
        char dSymTabNames[CDNUMREMEMBER][FILENAMESIZE];
        int dSymTabNumbers[CDNUMREMEMBER];
        int dNumSymbolTable;

        /*
         * Because CIF files may have FORWARD references, CDTo must
         * pass over the CIF file TWICE.  On the first pass,  it just
         * fills up the symbol name table.
         */
        int dFirstPass;

        /* True if debugging */
        int dDebug;
        int dNumSymbolsAllocated;
        }
    CDDesc;
```

The contents of the CD parameters descriptor *CDDesc* are available to all source files that include the *cd.h* header file. Most of the members in the CD parameters structure are used for the parsing or generating of CIF. They are used most frequently by the parser and action routines as working storage space for symbol information. See Chapter 3 that describes the CIF parser and action routines for a complete explanation of the use of each member in the CD parameters descriptor.

# Chapter 3

# The Fast CIF Parser

The CIF parser is the set of routines that interface KIC and CD to a standard intermediate layout language (CIF) and the secondary storage site for symbol definitions. This section describes the requirements of the layout language parser and the parameters that control its actions.

The term *fast* is used here to indicate that whenever there was a choice to be made between the size of the routines and their respective speed, the decision has always been to optimize for speed. Consequently, the parser is the largest module in KIC. With the optimizations that have been made to the parser, the *CDOpen()* routine is nevertheless limited by the speed of the parser that is impeded by the excessive overhead of memory allocation.

Three source files constitute the CIF parser; they are *parser.c*, *actions.c*, and *gencif.c*. The routines in *parser.c* scan the input file for the primitive commands of the layout language. The action routines are invoked by the parser when a primitive command is found. The gencif routines produce the primitives of the CIF layout language in the syntax that the parser understands.

Because the data model for the CD database is CIF, it would be difficult to replace the fast CIF parser with that of another layout language if the user should decide to do so. It is nevertheless possible to make CD *speak* in another language, given that the layout language is hierarchical and has similar geometry types. To accomplish this, the programmer would have to replace the source files *parser.c*, *actions.c*, and *gencif.c* as well as rewrite the CD routines *CDUpdate()*, *CDGenCIF()*, *CDTo()*, *CDFrom()*, and *CDParseCIF()*.

### 3.1. Action Routines

The function of the parser is to interpret the layout language, and the action routines are used by the parser to complete tasks that are specific to primitive commands of the layout language in one of three contexts. The context of actions is defined by the *dControl* member of the CD parameter descriptor *CDDesc* that can have one of four value that are defined in the *cd.h* file as follows:

```
/*
 * CD Control flags
 */
#define   DCONTROLCDOPEN     'o'
#define   DCONTROLPCIF       'p'
#define   DCONTROLCDTO       't'
#define   DCONTROLVANILLA    'v'
```

This section will describe the purpose of each of the above flags.

The action routines are invoked only by the parser. The DCONTROLVANILLA control character flag indicates that CD is not in the process of parsing CIF. An action routine should therefore never be invoked when the control flag is set to this value.

The DCONTROLCDTO control flag is used to signify that CD is currently translating a CIF file into individual KIC or CD symbol files. The action to be performed for this case is to output the primitive command that was found by the parser directly into the respective symbol file. If the *CDDesc.dRoot* flag is nonzero, output will be directed to the file referenced by *CDDesc.dRootFileDesc*, and output will otherwise be directed to the file referenced by *CDDesc.dSymbolFileDesc*.

The root file descriptor *CDDesc.dRootFileDesc* is initialized by *CDTo()* or whatever procedure that would invoke the parser with the *CDDesc.dControl* flag set to the value of DCONTROLCDTO. The symbol file descriptor *CDDesc.dSymbolFileDesc* however must be initialized by the action routine

*ABeginSymbol()* which is invoked for a new symbol definition, and therefore this action routine must be capable of determining the symbol name depending on the particular style of CIF that is identified by the value of *CDDesc.dProgram*. Also this routine must set *CDDesc.dRoot* to zero to direct subsequent output to the respective symbol file. The symbol file is closed and *CDDesc.dRoot* is set to unity by the action routine *AEndSymbol()* which is invoked at the termination of a symbol definition.

Because CIF may contain forward references to symbols, it is necessary for the action routines to build and maintain a symbol table. As a result, the parsing is a two-pass operation, where the first pass is dedicated to the construction of the symbol table by the *ABeginSymbol()* action routine. The symbol table is represented in the CD parameters descriptor as follows: the integer *dNumSymbolTable* is a count of the current symbols in the table. The character array *dSymTabNames*[*CDNUMREMEMBER*][*FILENAMESIZE*] contains the symbol names, and the array *dSymTabNumbers*[*CDNUMREMEMBER*] contains the corresponding symbol numbers.

All measurements are be scaled by the following value:

(CDDesc.dB * CDDesc.dDSA) / (CDDesc.dA * CDDesc.dDSB)

The integer values *CDDesc.dDSA* and *CDDesc.dDSB* are the symbol scale factors, dimensionless, and are currently always set to unity. The integer values *CDDesc.dA* and *CDDesc.dB* define respectively the micron per lambda scaling ratio. Because the conversion is from CIF, for which the database unit is one one-hundredth of a micron, to KIC or CD format, for which the database unit is one-hundredth of a lambda, the value of *CDDesc.dB* is in the numerator. The scale factor is computed for every metric value to avoid the use of floating point arithmetic; because integer arithmetic is significantly more fast than floating point, this is not a severe penalty.

The DCONTROLCDOPEN control flag is used when CD is currently parsing a symbol file. This action differs significantly from the previously described actions in that the CD database is constructed in local memory. When objects are discovered by the parser, the action routine for the object will insert it into the the symbol descriptor referenced by *CDDesc.dSymbolDesc* which is initialized by the *CDOpen()* procedure. Consequently, there is one action routine for each of the CD object creation procedures described in Section 2.4.4. There is no scaling performed on the objects when they are inserted into the symbol storage bins; metric data is represented in one one-hundredth lambda units in the KIC or CD symbol files.

Because the creation of an instance requires the invocation of several CD procedures using the same object pointer, the parameter *CDDesc.dPointer* is used as the object descriptor pointer for all CD object creation procedures. The property list referenced by *CDDesc.dPrptyList* as attached to each object or symbol after its creation. This allows property information to be saved in the CIF as user extensions.

The KIC or CD symbol files contain extensive information that allows the handling of forward references to be postponed until all geometric objects have been parsed, and thereby avoid two passes through the symbol file. When a symbol call is encountered, the action routine will insert the object descriptor for the instance into the storage bins and a reference is made in the master-list of the calling symbol. The *CDBeginMakeCall()* procedure will not attempt to open the instance in CD if the *CDDesc.dControl* flag is set to DCONTROLCDOPEN. When the parsing of a particular symbol has completed, *CDOpen()* will begin traversing the symbol's master-list, read all referenced symbols into the database if they have not already been opened, and invoke the *CDReflect()* procedure to reflect the bounding box of the every instance throughout the CD database. This algorithm allows *CDOpen()* to be called recursively to build a multi-level symbol hierarchy.

The *CDOpen()* routine will set the *CDDesc.dControl* control flag to the value of DCONTROLVANILLA before terminating.

The DCONTROLPCIF control flag is used when CD is constructing the database from a CIF file instead of a directory of KIC or CD symbol files. The actions to be performed are often identical to those for DCONTROLCDOPEN where the major differences are in the handling of symbol definitions. The symbol descriptor *CDDesc.dSymbolDesc* is initialized by *CDParseCIF()* or whatever procedure that would invoke the parser with the *CDDesc.dControl* flag set to the value of DCONTROLPCIF. The symbol descriptor *CDDesc.dRootFileDesc*, is used as temporarily storage of the *CDDesc.dSymbolDesc* symbol descriptor by the action routine *ABeginSymbol()* which is invoked when a new symbol definition is discovered by the parser. This action routine will then open a new symbol in the database for the new symbol definition and therefore this routine must be capable of determining the respective symbol name depending on the particular style of CIF that is identified by the value of *CDDesc.dProgram*. The descriptor pointer *CDDesc.dSymbolDesc* is again set to the pointer *CDDesc.dRootFileDesc*, by the action routine *AEndSymbol()* which is invoked at the termination of a symbol definition.

When objects are discovered by the CIF parser, the action routine for the object will insert it into the the symbol descriptor referenced by *CDDesc.dSymbolDesc*. There is no scaling performed on the objects when they are inserted into the symbol storage bins. An application program that uses the *CDParseCIF()* routine must be aware that *the size of the CD database unit is one one-hundredth of a micron.*

Because CIF may contain forward references to symbols, it is again necessary for the action routines to build and maintain a symbol table. As a result, the parsing is a two-pass operation, where the first pass is dedicated to the construction of the symbol table by the *ABeginSymbol()* action routine. The symbol table is represented in the CD parameters descriptor as follows: the integer *dNumSymbolTable* is a count of the current symbols in the table. The character array *dSymTabNames*[*CDNUMREMEMBER*][*FILENAMESIZE*] contains the symbol names, and the array *dSymTabNumbers*[*CDNUMREMEMBER*] contains the corresponding symbol numbers.

When the parsing of the CIF file has been completed, *CDParseCIF()* will traverse the master-lists of all symbols for the purpose of assuring that all referenced symbols are defined in the database and for bounding box propagation. The *CDBeginMakeCall()* procedure will not try to open the referenced symbol if the control flag *CDDesc.dControl* is set to DCONTROLPCIF, and the bounding box in the respective master-list descriptor is set to a null box having zero width and heigth. This is necessary because a referenced symbol may not have been inserted into the database at the time that the call command is recognized by the parser.

# Chapter 4

# The KIC User Interface

## 4.1. Window and Viewport Management

A window is an area in a world coordinate system that contains objects to be displayed. A viewport is the area on the graphics display in which the user views the contents of a window. In other words, the window defines the objects in the database to be displayed, and a viewport defines where to display the objects. The world coordinate system in KIC contains the CD symbol that is currently being edited. The unit of measurement in the world coordinate system is one one-hundredth of a lambda, the same unit used by CD.

A window may intersect an object in the current symbol such that the object is not contained entirely in the window, in which case the object would have to be *clipped* to the window before it is displayed in the viewport. For KIC, all windows and viewports are rectangular which simplifies the geometry clipping procedures.

The coordinate system in a window is identical to the world coordinate system. The coordinates in a viewport usually correspond to the resolution of the graphics display; this provides one viewport coordinate per display pixel. The origin of the coordinate system for the layout viewports in KIC is assumed to be the lower, left corner of the display. KIC uses a different coordinate system for the viewports that contain only textual information. In this textual coordinate system, a coordinate refers to a character block, and the origin is the upper right corner of the graphics display. For example, the text coordinate (1,1) refers to the graphical text character in the upper-most row and left-most column of the display; the text coordinate (5,10) refers to the graphical text character in the fifth row and tenth column of the graphics display.

KIC divides the graphics display into six viewports: the command menu viewport, the layer menu viewport, the information viewport, the prompt viewport, and the course and fine layout viewports. The following figure illustrates the relative positions of each viewport.

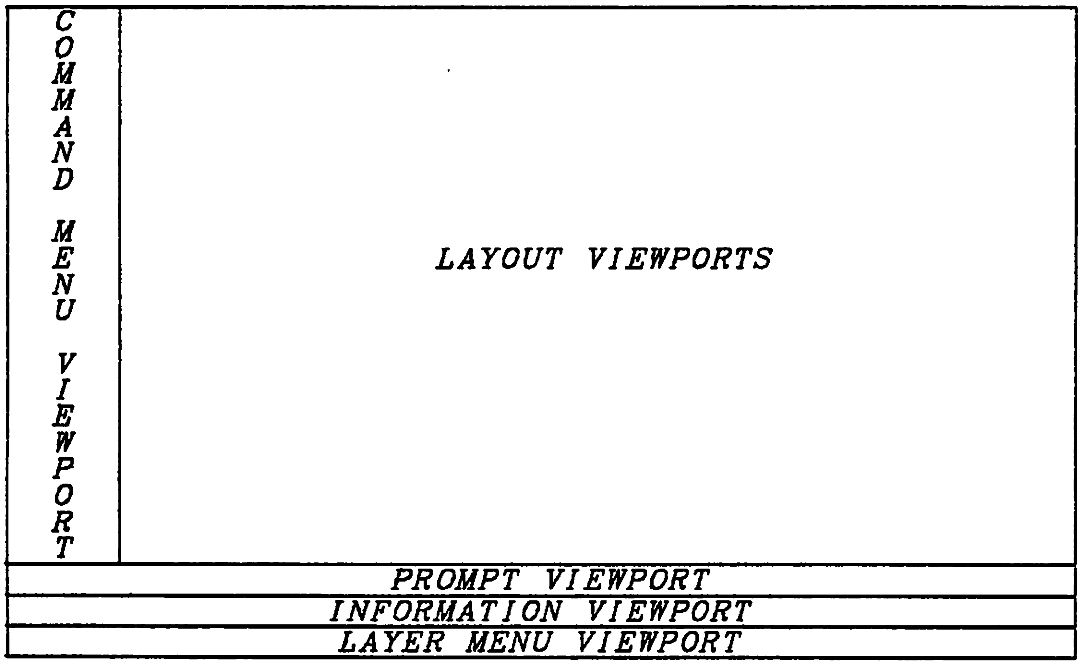

**Figure 2. The KIC viewports**

KIC represents windows and viewports with the area descriptor that is defined in the *kic.h* file as follows:

```
/*
 * Area structure
 */
struct ka {
     int kaLeft, kaBottom, kaRight, kaTop;
     int kaX, kaY;
     float kaWidth, kaHeight;
     };
```

See Section 4.2 that describes the basic KIC data structures for an explanation of the area structure members.

### 4.1.1. Text Viewports

There are four viewports used by KIC for displaying textual information; the layer menu viewport, the information viewport, the command menu viewport, and the prompt viewport. Because the position of the prompt viewport can be computed from the parameter viewport, the first three viewports are defined in the *kic.h* header file as follows:

```
/*
 * KIC text viewports
 */
struct ka MenuViewport;
struct ka LayerTableViewport;
struct ka ParameterViewport;
```

As described above, KIC uses a special coordinate system for viewports that display only textual information. A coordinate in this system refers to the space of one character block on the display given that no two characters overlap. The size and position of textual viewports in KIC are always represented by these character-block units, and only the *kaLeft*, *kaBottom*, *kaRight*, and *kaTop* members of the respective area structure are used to define a text viewport. By using this character block representation of graphic text, KIC simulates a typical ASCII character terminal.

#### 4.1.1.1. Layer Menu Viewport

The layer menu viewport is used by KIC to display the names and colors of all layers in the KIC or CD layer tables. The layer menu viewport always occupies the bottom text rows of the display, and is described first because the size and position of all other viewports depend on it's size. KIC will display in the layer viewport only those layers that are defined in the layer table, and therefore the size of the layer menu viewport is not fixed. KIC computes the number of layer names that can be displayed in a single row and saves this value in the *kpLayersPerMenuRow* member of the KIC parameters structure that is described in Section 4.3; a layer name is assumed to be less than or equal to four characters.

From the number of layers defined in the layer table, KIC then computes the number of text rows required for the layer menu viewport. These computations are performed by the *InitViewports()* procedure. After the viewports are initialized, the *ShowLayerTable()* procedure is invoked to display the layer table in the layer menu viewport.

The layer menu viewport is indeed a *menu* in that it is used to indicate and select the *current layer*. The current layer is represented by the *kpLayer* member of the KIC parameters structure and defines the layer to be used by layer dependent commands such as the geometry creation procedures. The current layer is indicated by a box around the name of the layer that is drawn by the *OutlineText()* procedure. When the KIC user points at a layer name in the layer menu viewport with the graphical pointing device, the *PointLayerTable()* procedure is invoked to determine the next current layer in KIC.

### 4.1.1.2. Parameter Viewport

The parameter viewport occupies the graphical text row that is immediately above the layer menu viewport. It displays current information such as the name of the current symbol and the width of the large, coarse window in lambda units. This information is displayed by invoking the *ShowParameters()* procedure. Also displayed in the parameter viewport is the lambda coordinate of the last point entered through the graphical pointing device and the displacement of this coordinate from the location of the previous pointing event. This information is displayed by invoking the *ShowXY()* routine.

### 4.1.1.3. Prompt Viewport

The prompt viewport occupies the graphical text row that is immediately above the parameter viewport and displays information that is relevant to the command procedure that the KIC user currently is executing. A character string is displayed in the prompt viewport by invoking the *ShowPrompt()* or

*ShowPromptAndWait()* procedures. The latter procedure will a bell character (control-G) to alarm the user of the prompt, and then wait for two seconds before continuing.

As it's name suggests, the prompt viewport is also used to prompt the KIC user for keyboard input. When the user responds to the prompt by typing at the keyboard of the graphics device, the characters that he types will be displayed (or echoed) in the prompt viewport. To inform the keyboard input routine of the character position at which to begin displaying input characters, the character size of every prompt string is saved by the *ShowPrompt()* routines in the *fLastCursorColumn* member of the current frame buffer structure that is described in Section 4.6.1.

#### 4.1.1.4. Command Menu Viewport

The command menu viewport is a textual viewport used by KIC to display the current command set or command menu. A command menu in KIC is represented by an array of character strings, each string containing the name of a particular command. When the KIC user points in the command menu viewport with the graphical pointing device, the string that is displayed on the row to which the user pointed is placed in the *kpCommand* buffer in the KIC parameters structure by the *Point()* procedure to identify the selection of a command. The *Point()* procedure is described further in Section 4.5.

The width of the command menu viewport is exactly five columns or five character block units, which requires command names to be no longer than five characters. The left edge of the command menu viewport is the is the left edge of the graphics display, and the top of the viewport is also the top of the display. The bottom row of the command menu viewport is the row that is immediately above the prompt viewport.

A command menu is displayed in the command menu viewport by the

*ShowMenu()* procedure. When a menu command is selected by the KIC user, it is highlighted by the *MenuSelect()* procedure.

### 4.1.2. Layout Viewports

There are two windows in KIC that will map to one of three viewports for displaying layout information; the coarse viewport, the large coarse viewport, and the fine viewport. No more than two layout viewports are ever used at any given time. The layout windows and viewports are defined in the *kic.h* file as follows:

```
/*
 * Windows and the viewports they map to.
 */
int FineViewportOnBottom;
int FineWindowWidth,FineWindowHeight;
struct ka CoarseViewport,CoarseWindow;
struct ka LargeCoarseViewport,SmallCoarseViewport;
struct ka FineViewport,FineWindow;
```

The coarse window *CoarseWindow* is typically used to define the general area in which the KIC user is working, and the fine window *FineWindow* defines a smaller working area with greater resolution. The fine window is generally contained in the area of the coarse window, but is not constrained to be such.

The four defined viewports are allocated as follows: the large coarse viewport *LargeCoarseViewport* represents the entire area of the screen that is dedicated for displaying layout information. The fine viewport represents either the bottom third or the left half of the layout area depending on the logical value of *FineViewportOnBottom*. After the display area of the fine viewport has been allocated, the remaining layout area is represented by the small coarse viewport *SmallCoarseViewport*. The current coarse viewport, the one that will be displayed depending on the width of the coarse window, is represented by the descriptor *CoarseViewport*. Unlike text viewports, the layout viewports are measured by numbers of display pixels. The size of the large and small coarse viewports, the fine viewport, and all text viewports is computed by the

*InitViewport()* procedure.

The coarse window is displayed in either the large coarse viewport or the small coarse window depending on the width of the window. When the number of pixels per lambda in the large coarse viewport exceeds roughly the value of *kpPointingThreshold* in the KIC parameters structure, only the large coarse viewport is displayed, and the fine viewport is not displayed. For larger windows, the contents of the coarse window are displayed in the small coarse window, and the fine window is displayed in the fine coarse viewport. The decision of whether the coarse window is displayed in the large or small coarse viewport is made in the *SwitchToFinePositioning()* procedure.

To generalize, there are two modes in KIC for displaying layout information; one mode is to display only the coarse window in the large coarse viewport, and the second mode is to display the coarse window in the small coarse viewport and to display the fine window in the fine viewport. The current mode of display can be determined from the contents of *kpRedisplayControl* in the KIC parameters structure that can assume one of following values defined in *kic.h* file:

```
/*
 * Viewport control flags
 */
#define   SPLITSCREEN            'b'
#define   FINEVIEWPORTONLY       'f'
#define   COARSEVIEWPORTONLY     'c'
```

If both the fine and small coarse viewports are displayed, the *kpRedisplayControl* parameter will be assigned the value of SPLITSCREEN by default, and the *kpDisplayFineViewport* parameter will be set to the value of *True*. If only the large coarse viewport is displayed, then *kpRedisplayControl* will assume the value of COARSEVIEWPORTONLY by default, and the *kpDisplayFineViewport* parameter will be set to the value of *False*. By setting the display control parameter to the value of FINEVIEWPORTONLY, only the fine viewport will be updated by a display routine, and the COARSEVIEWPORTONLY switch will result in only the small or large coarse viewport to be effected by a display routine. Any procedure that uses the display control parameter to control the KIC geometry display routines must reset the parameter to it's default value before terminating.

### 4.1.3. Clipping

Because KIC was written to run on most "dumb" or low performance graphics terminals, it is necessary to accomplish window-to-viewport geometry clipping on the host machine instead of down-loading the task onto the graphics device. Clipping is the procedure by which sections are removed from a particular geometry such that the contour will be contained entirely in the targeted viewport. Because KIC uses only rectangular viewports, the clipping of rectangles is trivial and will not be explained here in detail. The *LToP()* macro, which converts lambda coordinates to display coordinates, performs a rectangular clipping to the destination viewport.

If the graphics device provides such capabilities as geometry clipping and definable viewports and windows, these tasks can be down-loaded to the device, but probably not without considerable rewriting of code. The graphics device would be required to support a world coordinate system that is as extensive as the system used by CD. Many terminals that claim to have this ability limit the world coordinate values to short integers that are inadequate for an IC layout with sub-lambda resolution. The KIC window management system may have to be modified to be efficiently adapted to graphics device with definable viewports. At present, the KIC geometry display routines operate under the assumption that objects can be displayed in any layout viewport at any time; in other words, there is no notion of a current viewport. Considerable overhead may result from viewport context switching on a graphics device that handles the KIC viewport management in this manner.

The procedure *MFBPolygonClip()* for clipping polygons to a window is provided by the MFB library of graphics routines and is described in detail in [6]. It is an easy task to clip a line segment to a half-plane when the edge of the half-plane is parallel to a coordinate axis. If we consider the interior of a window or viewport to be the intersection of four such half-planes, the clipping of a polygon can be performed by traversing the edge list of the polygon once for each half-plane, clipping the edges to the respective half-plane.

Polygon clipping is performed in the world coordinate system in KIC; a polygon is clipped to a particular window and then mapped to the targeted viewport. KIC invokes the frame buffer interface routine *FBPolygonClip()* which then invokes the *MFBPolygonClip()* procedure. This provides the programmer with the ability to easily insert his own polygon clipping routine, if he wishes to, and still use routines in the MFB graphics library.

### 4.1.4. Window/Viewport Transformations

KIC uses the *LToP()* macro for converting from lambda database coordinates to pixel display coordinates, and the *PToL()* procedure converts from display coordinates to lambda coordinates in a given window; the macro is used for speed and performance considerations. For a more complete explanation of window/viewport transformations, see [6].

The *LToP()* macro performs window-to-viewport clipping of rectangles and is defined as follows in the ***coords.h*** header file:

```
#define LToP(Viewport,Window,X,Y){                                 \
        X = (int)(((float)(X-Window.kaLeft)*Viewport.kaWidth)      \
                          /Window.kaWidth)+Viewport.kaLeft;         \
        Y = (int)(((float)(Y-Window.kaBottom)*Viewport.kaHeight)   \
                          /Window.kaHeight)+Viewport.kaBottom;      \
        if(X < Viewport.kaLeft) X = Viewport.kaLeft;               \
        else if(X > Viewport.kaRight) X = Viewport.kaRight;        \
        if(Y < Viewport.kaBottom) Y = Viewport.kaBottom;           \
        else if(Y > Viewport.kaTop) Y = Viewport.kaTop;            \
        }
```

The efficiency of the procedure is essential because it must be executed at least twice before any geometry can be displayed on the graphics device. Floating point arithmetic is however used for both safety and accuracy. If the computation did not use floating point arithmetic and assuming one hundred database units per lambda, the approximate allowable window width before an overflow occurred would be fourty thousand lambda (for a 1k by 1k display resolution and a four byte integer representation). If forty thousand lambda is an acceptable size for a world coordinate system, it is recommended that the transformation use only integer arithmetic for efficiency; experience has frequently shown, however, that a larger world coordinate system is desirable.

The *PToL()* macro performs viewport to window coordinate conversion and also performs cursor snapping. Cursor snapping is the procedure by which coordinates that are returned from the graphical pointing device are constrained to lie on a grid. The lambda spacing between adjacent points on the grid is specified by the *kpPixToLambdaSnapping* member in the KIC parameters structure defined in Section 4.3. *PToL()* is defined in the *coords.h* file as follows:

```
#define   HALFSNAPPING   Parameters.kpHalfPixToLambdaSnapping
#define   SNAPPING       Parameters.kpPixToLambdaSnapping

PToL(Viewport,Window,X,Y)
    struct ka Viewport, Window;
    int *X, *Y;
    {
    float tmp1, tmp2;
    tmp1 = ((float)(*X - Viewport.kaLeft));
    tmp2 = Window.kaWidth / Viewport.kaWidth;
    *X = ((int)(tmp1 * tmp2)) + Window.kaLeft;
    if(*X >= 0) *X = ((*X+HALFSNAPPING)/SNAPPING)*SNAPPING;
    else *X = ((*X-HALFSNAPPING)/SNAPPING)*SNAPPING;
    tmp1 = ((float)(*Y - Viewport.kaBottom));
    tmp2 = Window.kaHeight / Viewport.kaHeight;
    *Y = ((int)(tmp1 * tmp2)) + Window.kaBottom;
    if(*Y >= 0) *Y = ((*Y+HALFSNAPPING)/SNAPPING)*SNAPPING;
    else *Y = ((*Y-HALFSNAPPING)/SNAPPING)*SNAPPING;
    }
```

The *PToL()* procedure is invoked whenever the KIC user points in a viewport with the graphical pointing device or when KIC is required to compute the size in lambda of an area on the graphics display; efficiency is certainly not an issue. Because it is essential, however, that the *PToL()* procedure be the inverse of *LToP()*, accuracy is an issue, and, therefore, floating point arithmetic is used. If one procedures was not the inverse of the other, the integrity of the layout display and graphical pointing device would be questionable, and that for a graphics editor would be intolerable.

### 4.2. KIC Data Structures

The basic data structures of KIC are described in this section. The KIC parameters structure is described separately in Section 4.3.

#### 4.2.1. The Area Descriptor

The KIC area descriptor is defined in the *kic.h* header file as follows:

```
/*
 * Area structure.
 */
struct ka {
        int kaLeft, kaBottom, kaRight, kaTop;
        int kaX, kaY;
        float kaWidth, kaHeight;
        };
```

The KIC area structure is used for storing the representations of both rectangles and rectangular areas. Every possible representation of the rectangle is contained in this structure for convenience. The *kaLeft*, *kaBottom* members define the untransformed lower, left coordinate of the rectangle, and the *kaRight*, *kaTop* members define the untransformed upper, right coordinate of the rectangle. The *kaX*, *kaY* members represent the untransformed center of the rectangle, and the *kaWidth* member defines the width of the rectangle in the horizontal direction, and *kaHeight* defines the height of the rectangle in the vertical direction.

The area structure is also used by KIC for representing the several windows and viewports that are described in Section 4.1. In practice, the *kaWidth* and *kaHeight* members are used exclusively for this purpose if and only if the viewport or window is used for the display of layout information. They are defined as floating point values so that the window-to-viewport transformation routines, which use floating point arithmetic for accuracy, are not required to compute the floating point width and height and for every window-to-viewport coordinate transformation.

The *kaWidth* and *kaHeight* structure members are never used when the *ka* area structure represents a rectangle or one of the four viewports that display only textual information; they are used exclusively for layout viewports or windows.

### 4.2.2. The Window Stack Descriptor

The KIC window structure is defined as follows in the *kic.h* header file:

```
/*
 * Structure used to save windows in window stack
 */
struct kw {
     int kwLastWindowX;
     int kwLastWindowY;
     int kwLastWindowWidth;
     int kwLastFineWindowX;
     int kwLastFineWindowY;
     char kwName[8];
     };
```

KIC provides the user with the ability to assign names to specific windows and to save the respective window on a stack such that the user can randomly return to any desired view of the layout. An array of *kw* structures is just such a list of layout windows. The *kwName* character string is the user-specified name for the window. The structure members *kwLastWindowX* and *kwLastWindowY* define the center coordinate of the window, and *kwLastWindowWidth* defines the width of the coarse window in one one-hundredth lambda units. The coordinate *kwLastWindowX*, *kwLastWindowY* defines the center of the fine window or magnifying glass.

The width of the fine window is assumed to be that default value that is computed by the *InitFineWindow()* initialization routine. If the width of the coarse window is such that it is not necessary to display the fine window, this condition would be recognized by invoking the *SwitchToFinePositioning()* procedure after retrieving a window definition from the window stack.

### 4.2.3. The KIC Layer Table Descriptor

The KIC layer descriptor and layer table are defined in the *kic.h* header file as follows:

```
/*
 * The following information is read from the .KIC file.
 */
struct kl {
     int klR, klG, klB;              /* RGB color */
     int klMinDimensions;            /* Minimum lambda dimensions */
     int klFilled;                   /* filled or outlined? */
     int klStyle[8];                 /* bit array for fill pattern */
     int klStyleID;                  /* style ID */
     int klCoarseStyleID;            /* style ID for Coarse window */
     int klFineStyleID;              /* style ID for Fine window */
     int klVisible;                  /* visibility */
     int klBlink;                    /* blinking layer? */
     int klSymbolic;                 /* symbolic? */
     int klWireWidth;                /* wire width >= mindimensions */
     char klTechnology;              /* layer name */
     char klMask[3];
     }
LayerTable[CDNUMLAYERS+1];
int NumLayerTable;
```

An array of KIC layer descriptors represents the KIC layer table that differs from the CD layer table in that it contains display information as well as layout guidelines. The size of the KIC layer table is identical to that of the CD layer table, and the entries correspond to directly the CD layer numbers. The name of a particular layer represented by a KIC layer descriptor is given by the character string *klTechnologyklMask* and also is identical to the name of the corresponding layer in the CD layer table. As in the CD database, layer zero represents the instance layer, and because KIC has special procedures for displaying instances, the KIC layer descriptor for layer zero is unused.

The *klMinDimensions* structure member specifies the minimum lambda dimension for the specific layer in the given process technology. If the user creates a wire in the layout, the width of the new wire will be *klWireWidth* lambda units, where *klWireWidth* is greater than or equal to the value of *klMin-*

*Dimensions*. The *klSymbolic* member identifies the particular layer as either a symbolic representation of data or a mask level.

The *klR*, *klG*, and *klB* structure members define the RGB color combination with which the layer is to be displayed in the layout viewports. All color intensities are normalized by KIC to 255.

KIC displays layers in the layout viewports as either filled or outlined. If a particular layer is to be filled when displayed, it can have a fill pattern associated with it; the outlining of a layer is not considered as a fill pattern in the KIC display philosophy. The *klFilled* member in the KIC layer descriptor is a boolean that specifies whether the respective layer should be filled or outlined when displayed. The *klStyleID* member is the index that identifies the fill pattern for the respective layer; the value zero is always assumed to represent a solid-fill pattern. The *klStyle* array contains and eight by eight intensity array that defines of the fill pattern that is attributed to the respective layer. The eight least significant bits of each integer in the *klStyle* array are used to represent a row of the pattern.

Because a graphics terminal can in general display solid filled objects more rapidly than stippled or pattern filled objects, KIC uses a thresholding procedure for displaying filled geometries in the layout viewports to decrease the time required to display a window. To do accomplish this, there are separate indices to identify the fill pattern for a particular layer in the fine and coarse layout viewports. The number that identifies the fill pattern for a layer in the coarse viewport is *klCoarseStyleID*, and *klFineStyleID* is the fill pattern index for the fine layout viewport; for both indices, the value of zero identifies a solid-fill pattern, and layers are always outlined if the *klFilled* layer descriptor member is zero, regardless of the value of the fill pattern index. If in either layout viewport the number of pixels that are required to display the length of one lambda unit is less than two, the fill pattern index for the viewport is set to zero to force a solid-fill pattern. This decision is made in the *InitVLT()* routine that must therefore be invoked whenever the size of a viewport or window changes.

The remaining descriptor members are used specifically as display controls. If the *klBlink* member is set for a particular layer, the layer will be displayed in a blinking mode on the graphics terminal, given that the device has the capability of blinking colors. If the *klVisible* member is zero, the respective layer will not be displayed in the layout viewports; that is, it will be invisible.

### 4.2.4. The Cursor Descriptor

The KIC cursor descriptor is defined in the *kic.h* file as follows:

```
/*
 * Cursor desc.
 */
struct kc {
      /* In lambda units.  */
      int kcPredX, kcPredY, kcX, kcY;
      int kcDX, kcDY;
      int kcRow, kcColumn;
      }
Cursor;
```

The cursor descriptor is used by the *Point()* routine in KIC to report a user pointing event. When the KIC user points in a layout viewport using the graphical pointing device, the lambda coordinate of the user-selected point is placed in the *kcX*, *kcY* structure members of the KIC cursor descriptor by the *Point()* procedure. The previous user-selected point is moved from the *kcX*, *kcY* structure members to *kcPredX*, *kcPredY*, and the orthogonal displacements between the two lambda coordinates are computed and placed in *kcDX*, *kcDY*. Whenever the KIC user points in a layout viewport, the *kcRow* and *kcColumn* descriptor members are set to zero.

When the KIC user points in a text viewport using the graphical pointing device, the respective row-column text coordinate is placed in the *kcRow*, *kcColumn* structure members of the KIC cursor descriptor by the *Point()* procedure. As described in Section 4.1.1, the row-column text coordinate system divides the graphical display into a grid of graphical character blocks with the origin being in the upper, right corner of the display.

After the *Point()* routine is invoked in KIC, KIC determines the viewport to which the user pointed by testing the values of certain integer flags in the KIC parameters structure that is described in Section 4.3. The parameters of interest here are *kpPointCoarseWindow* and *kpPointLayerTable.*

### 4.2.5. The KIC Selection Queue

The KIC selection queue is defined in the *select.h* header file as follows:

```
struct ks {
     struct ks *ksSucc;
     struct o *ksPointer;
     };
struct ks *SelectQHead;
struct ka SelectQBB;
```

The selection queue is a linked-list of *ks* structures that is used by KIC to identify specific objects in a particular symbol. The list of objects is typically used to identify a set of objects that will be subject to an operation such as move or copy. The *SelectQHead* pointer references the first *ks* structure in the linked-list and is a null pointer if the list is empty. When a an object is inserted into the selection queue, the respective object descriptor pointer is saved in the *ksPointer* member, and the *ksSucc* pointer member is assigned to the value of the *SelectQHead* pointer. The *SelectQHead* pointer is then set to point to the new member in the selection queue. In other words, new items in the selection list are always inserted at the head of the list.

When the *SelectQComputeBB()* procedure is invoked, the bounding box of all objects in the selection queue is computed and stored in the *SelectQBB* area structure.

### 4.2.6. The Context Descriptor

The KIC context descriptor is defined in the *contexts.c* source file as follows:

```
#include "cd.h"
/*
 * Context stack shouldn't get deeper than transformation stack.
 */
struct cc {
     int ccX,ccY,ccWidth,ccModified;
     int ccNumWindows;
     struct kw ccSaveWindow[VIEW_STACK_SIZE];
     struct o *ccInst;
     char ccMaster[81];
     }
Context[XFORMSTACKSIZE];
int ContextSP = 0;
```

The context stack is used by KIC to save the context of an editing session while the KIC user edits another symbol. A typical scenario is as follows: the user is editing a symbol with an instance of another symbol. The user discovers that the called symbols is, perhaps, lacking a contact. He then places the contexts of the current editing session in the context stack and begins editing the defective symbol. When the problems in the called symbol have been corrected, the user retrieves the previous editing session from the context stack and begins editing where he was before he discovered the error in the instance.

The size of the context stack can not be larger than that of the transformation stack because the transformations that are associated with the symbols being edited are saved on the transformation stack. The name of the symbol that is being edited in the respective context is contained in the *ccMaster* character string, and *ccInst* is a pointer to the object descriptor of the instance that the KIC user began editing after the context was placed on the context stack. The *ccX*, *ccY*, and *ccWidth* define the size and position of the coarse window; the size and position of the fine window is assumed to be the default value computed by the *InitFineWindow()* procedure. The window stack that is associated with the respective editing context is defined by the *ccNumWindows* and *ccSaveWindow* structure members.

The procedures for context switching are appropriately called *Push()* and *Pop()*. Both routines are responsible for maintaining the context stack pointer *ContextSP*. The *Push()* routine assumes that the object that is referenced by the first item in the selection queue is the called symbol to become the next editing context.

### 4.3. The KIC Parameters Structure

The KIC parameters structure contains the controlling parameters of the KIC program that may be shared by all program modules. Because this data structure is so large and the members mostly unrelated, this section may be the most confusing of any section of this document. Nevertheless, a thorough knowledge of the internal structure of KIC would require the programmer to recognize all parameter structure members.

The KIC parameters structure is defined in the *kic.h* file as follows:

```
struct kp {
        /* Symbol desc for current cell */
        struct s *kpCellDesc;

        /*
         * Object desc for a geometry currently being created.
         * KIC special cases the input of polygons and wires.
         */
        struct o *kpPointer;

        /* True if instances should be expanded */
        int kpExpandInstances;

        /* True if instance is expanded in fine viewport only */
        int kpExpandFineViewportOnly;

        /* If False then the SelectQ is not redisplayed */
        int kpEnableSelectQRedisplay;

        /* Color ID's for command menu */
        int kpMenuTextColor;
        int kpMenuHighlightingColor;
        int kpMenuSelectColor;

        /* If True, user pointed to layer table and Command[0] == EOS */
        int kpPointLayerTable;

        /* If True, user pointed to coarse viewport and Command[0] == EOS */
        int kpPointCoarseWindow;

        /* Control of the Layer Menu */
        int kpNumLayerMenuRows;
        int kpLayersPerMenuRow;

        /* Number of sides for round flashes */
        int kpNumRoundFlashSides;

        /* Current layer */
        int kpLayer;

        /* True if selection commands are LayerSpecific */
        int kpLayerSpecificSelection;
```

```
/* If True, then outline all stippled geometries */
int kpOutline;

/* If True, polygon vertices are clipped to nearest grid point */
int kpClipVerticesToGrid;

/* If True, put grid below layout geometries */
int kpGridOnTop;

/* If True, grid will be shown in large viewport */
int kpShowGridInLargeViewport;

/* Color ID's for grid */
int kpCoarseGridColor;
int kpFineGridColor;

/* Number of RESOLUTION*lambda between grid points. */
int kpGrid;

/* True if current cell has been modified */
int kpModified;

/* Parameters for modifying geometries */
int kpModifyLeft;
int kpModifyTop;

/* Bounds of coordinate system */
int kpMaxX, kpMaxY, kpMinX, kpMinY;

/* Debug parameters */
int kpNumGeometries;

/* If True, then show redisplay bandwidth */
int kpShowBandwidth;

/* If True, user has just pressed the interrupt key    */
int kpSIGINTERRUPT;

/*
 * == COARSEVIEWPORTONLY if only coarse viewport should be displayed
 * == FINEVIEWPORTONLY if only fine viewport should be displayed
 * == SPLITSCREEN if both should be displayed
 */
int kpRedisplayControl;

/* If True, Fine Viewport (Magnifying Glass) is displayed */
int kpDisplayFineViewport;

/* If True, all text is displayed */
int kpDisplayAllLabels;

/* If True, all instances will be labeled in the viewport */
int kpLabelAllInstances;

/* If True, instances will be marked when selected */
int kpShowInstanceMarkers;

/*
 * PointingThreshold is the minimum value of ViewportWidth/WindowWidth
 * such that it is still comfortable to point with lambda precision.
 */
int kpPointingThreshold;
```

```
          /* True if wires and polygons should be constrained to 45s */
          int kp45s;

          /*
           * PixToLambdaSnapping is RESOLUTION times the number of lambda
           * between points to which a cursor input point is snapped.
           */
          int kpPixToLambdaSnapping;
          int kpHalfPixToLambdaSnapping;

          /* Current transform defined in Selection menu */
          int kpRotationAngle;/*0, 90, 180, or 270 */
          int kpMX;
          int kpMY;

          /* At what level in the hierarchy are we?  See Redisplay */
          int kpHierarchyLevel;

          /*
           * Window stack.  kpWindowStack[0] is always the last view.
           * If kpNumWindows is zero, only the last view is saved.
           */
          struct kw kpWindowStack[VIEW_STACK_SIZE];
          int kpNumWindows;

          /* Background and Highlighting color control */
          int kpHighlightingPixel;
          int kpHighlightingRed;
          int kpHighlightingGreen;
          int kpHighlightingBlue;
          int kpBackgroundRed;
          int kpBackgroundGreen;
          int kpBackgroundBlue;

          /* used in Attributes.c */
          int kpSetBackgroundColor;
          int kpSetHighlightingColor;

          /* Symbol name for current cell */
          char kpCellName[80];

          /* Command selected if any from command menu */
          char kpCommand[80];

          /*
           * Current command menu
           *          == INSTANCEMENU denotes instance menu
           *          == ATTRIBUTESMENU denotes attribute menu
           *          == PROPERTYMENU denotes property menu
           *          == BASICMENU denotes basic menu
           *          == SELECTIONMENU denotes selection menu
           *          == DEBUGMENU denotes debug menu
           *          == AMBIGUITYMENU denotes ambiguity menu
           */
          char kpMenu;

          }
     Parameters;
```

The KIC parameters structure contains most of the controlling variables of the KIC program as well as the information that is shared between the separate KIC procedures. There is at least one mechanism in KIC to allow the user to modify each of the structure members.

The name of the symbol that is currently being edited by KIC is saved in the *kpCellName* character buffer, and the respective CD symbol descriptor is referenced by the *kpCellDesc* member. The current symbol descriptor is set in the *Edit()* procedure only.

Another CD descriptor in the KIC parameters structure is the *kpPointer* object descriptor pointer that is used to reference an object that is being inserted into the current CD symbol. The display of a polygon or wire that is currently being created in the layout is given special consideration; the object will be displayed above all other objects so as not to obscure the detail. If the user redefines the position of the fine window while in the process of specifying the contour of the wire or polygon, the *ShowFineWindow()* routine will recognize that the pointer is not a null pointer and display the referenced object above all other objects in the window if the respective object intersects the fine window.

The *kpCommand* character buffer indicates when a user has selected a menu command and identifies what the specific command is. When a user points at a command in the menu viewport with the graphical pointing device, the name of the command is placed in this character buffer by the *Point()* routine, or more specificly by the *CtrlAt()* procedure. If the user types at the keyboard of the of the graphics terminal while the graphical pointing device is active, the character that the user types is placed at the end of the contents of this buffer by the *Point()* procedure. If the user does not point in the command menu viewport or does not type at the terminal keyboard, the *Point()* routine returns with the first character of the *kpCommand* buffer set to the value of *EOS*. Any procedure that calls the *Point()* routine is required to test the contents of this buffer to allow the KIC user the opportunity to make use of the command menu at all times.

There are two other parameter structure members that are set by the *Point()* routine only. The *kpPointLayerTable* integer is set to the logical value of *True* if the KIC user points in the layer menu viewport with the graphical pointing device, and the *kpPointCoarseWindow* is set to the value of *True* if the user points in the layout viewports, regardless of whether the fine window intersects the coarse window. Both of the structure members are by default set to the value of *False* by the *Point()* routine. If the user points in the layer menu viewport, the *PointLayerTable()* procedure sets the *kpLayer* structure member to the value of the index of the new current layer in the KIC layer table.

Several parameters that control the *Redisplay()* routine are *kpExpandInstances*, *kpExpandFineViewportOnly*, *kpEnableSelectQRedisplay*, and *kpShowBandWidth*. If the *kpExpandInstances* integer flag is set to the logical value of *True*, all instances will be displayed in full detail in both layout viewports; otherwise, the instances will be represented in the layout viewports by only a line around the perimeter of the bounding box of the cell. If the *kpExpandFineViewport* integer is set to the logical value of *True*, all instances will be expanded in full detail in the fine viewport only; an instance in the coarse viewport will again be represented by the contour of its bounding box. If the *kpEnableSelectQRedisplay* integer is set to the logical value of *True*, the *Redisplay()* routine will invoke the procedure *SelectQShow()* before terminating; this latter procedure will highlight in both layout viewports all objects contained in the selection queue. The *kpNumGeometries* integer is set to zero when the *Redisplay()* procedure is invoked and then incremented whenever a geometric object is displayed thereby providing a count of the number of geometries. If the *kpShowBandWidth* integer flag is set, the *Redisplay()* routine will compute the real, user, and system time required to display the particular area in the layout viewports and report these values to the user before terminating.

The *Redisplay()* routine will invoke the *ShowGrid()* procedure that displays a grid in the layouts viewports according to the user-specified options. If the *kpGrid* integer member of the parameters structure is less than unity, then a grid will never be displayed in the layout viewports; otherwise, the value of *kpGrid* specifies the lambda spacing of the grid lines starting at the origin of the world coordinate system. If the number of display pixels per lambda in the fine viewport exceeds the value of the *kpPointingThreshold* parameter, then a grid will be displayed in that layout viewport, and if the *kpShowGridInLargeViewport* parameter is set to the logical value of *True*, then a grid will also be displayed in the large coarse layout viewport if the display pixel per lambda ratio is large enough to permit a grid. The *Redisplay()* procedure will invoke the *ShowGrid()* procedure after all layout geometries have been displayed if the *kpShowGridOnTop* parameter is set to the logical value of *True* that will result in the lines of the grid being displayed over layout geometries. If the *kpShowGridOnTop* parameter is not set, the grid will be displayed before layout geometries that will cause the view of grid lines to be obstructed by layout geometries.

The lines of the grid that is displayed by the *ShowGrid()* procedure will be displayed in two colors. The *kpFineGridColor* parameter is the index of the layer in the KIC layer table that will be used to display the intermediate grid lines. Every fifth line of the grid appear in the color of the layer that is indexed in the KIC layer table by *kpCoarseGridColor*.

The parameters structure contains other viewport color information. The *kpMenuTextColor* is the index of the layer in the KIC layer table whose color will be used as the color of all graphic text in the textual viewport. The *kpMenuSelectColor* and *kpMenuHighlightingColor* members of the parameters structure define the color indices with which all graphic text will be displayed and highlighted in the command menu viewport if the respective command is selected; a highlighted command means that the name of the command is written over a box that is displayed in the color of the layer that is indexed in the KIC layer table by *kpMenuHighlightingColor*.

The *kpBackgroundRed*, *kpBackgroundGreen*, and *kpBackgroundBlue* parameter structure members define the red-green-blue combination of the background color for all viewports. The background color for KIC is always the first color in the video color table of the graphics terminal. The *kpHighlightingRed*, *kpHighlightingGreen*, and *kpHighlightingBlue* parameters define the red-green-blue combination of the highlighting color. An object is highlighted in the layout viewports by displaying a line in the highlighting color around the perimeter of the object. The highlighting color for KIC is always the last color in the video color table of the graphics terminal and is always indexed by the *kpHighlightingPixel* parameter structure member.

Other members of the parameters structure that control the display of information in the layout viewports are as follows: the *kpOutline* parameter is set to the value of true if stippled geometries are to by outlined; depending on the fill patterns being used, it is occasionally more pleasing to the eye to have the perimeter of stipple filled geometries clearly visible. If the *kpOutline* flag is set, however, the time that is required for the graphics terminal to redraw the layout viewports will of course increase. Because KIC assumes that the graphics terminal is not capable of scaling graphic text, the *ShowLabel()* routine, which is invoked by the *Redisplay()* routine to display a label in the layout viewports, will not display a label in a layout viewport if the number of display pixels per lambda is less than two; the number two was simply chosen after trying several other values and finding that two was most reasonable for typical IC layouts. If the *kpDisplayAllLabels* parameter is set to the logical value of *True*, the *ShowLabel()* routine will always display labels in the layout viewports regardless of the window-to-viewport scale factor. If the *kpShowInstanceMarkers* member of the KIC parameters structure is set, the *SelectQShow()* procedure, which is invoked by the *Redisplay()* procedure to highlight the content of the selection queue, will display a small, diamond shaped marker at the reference point of each instance in the selection queue; the reference point of an instance is also the origin of the world coordinate system for the respective symbol and the point about which the cell would be rotated or mirrored.

The procedure that places objects into the selection queue is controlled by the *kpLayerSpecificSelection* member of the KIC parameters structure. If this parameter is set when KIC asks the user to point in the layout viewports with the graphical pointing device to identify the object to be placed in the selection queue, only those objects that lie on the current layer, as defined by the *kpLayer* parameter, will be placed in the selection queue. If under this condition of layer specific selection, the KIC user does not point to any object in the layout viewports that is on the current layer but does point to an instance, the respective instance will be placed in the selection queue. If the *kpLayerSpecificSelection* parameter is set to the logical value of *False*, all objects that the KIC user identifies by pointing in the layout viewports will be placed in the selection queue.

There are several parameters in the KIC parameters structure that control the creation and editing of objects by KIC. When the KIC user creates a wire or polygon and begins to define the contour of the object, the angle formed by any two adjacent segments along contour will be constrained to integer multiples of

45 degrees if the *kp45s* member of the parameters structure is set to the logical value of *True*. The *kpRotationAngle*, *kpMX*, and *kpMY* parameters define the current transformation; if a new instance is placed in the symbol or an object is moved or copied, the operation is performed with the transformation that is defined by these parameters. If the *kpMX* parameter is set, the transformation will mirror in the direction of the x axis, and if the *kpMY* is set, the current transformation will mirror in the direction of the y axis. The current rotation angle *kpRotationAngle* may only be set to either the value of 0, 90, 180, or 270 degrees. The *kpModifyLeft* and *kpModifyTop* parameters are used exclusively to specify the direction in which the KIC user will stretch a rectangle using the KIC stretch-box command. If the *kpModifyLeft* parameter is set to the logical value of *True*, then the left edge of the rectangle will be stretched; otherwise, the right edge will be stretched. As explained in Section 2.4.4, KIC does not use the CD round flash descriptor, preferring instead to represent round objects as polygons, including arcs and doughnuts. If the KIC user creates a round flash or doughnut, the *kpNumRoundFlashSides* integer will specify the number of sides on the outer perimeter of the polygon that will represent the round object. KIC will not allow the value of *kpNumRoundFlashSides* to be less than eight and not greater than 360. If the *kpClipVerticesToGrid* parameter is set, the vertices of all polygonal objects will be clipped to the lambda grid; this could result in peculiarly shaped round flashes or doughnuts if the respective diameter is small. Finally, if the user creates a new object in the current symbol, or modifies an existing one, the *kpModified* flag is set to the value of *True*, which will cause KIC to remind the user if the user attempts to abort or edit a new symbol without updating the current one.

The KIC parameters structure contains information that is used to control many viewport actions. The *kpMenu* member of the parameters structure defines the command menu that is currently displayed in the command menu viewport. The possible values for the *kpMenu* parameter are defined in the *kic.h* file as follows:

```
/*
 * Menu names
 */
#define   BASICMENU          'b'
#define   DEBUGMENU          'd'
#define   SELECTIONMENU      's'
#define   INSTANCEMENU       'i'
#define   ATTRIBUTESMENU     'a'
#define   PROPERTYMENU       'p'
#define   AMBIGUITYMENU      'A'
```

The number of layers that can be displayed in a single row of the layer menu viewport is defined by the value of the *kpLayersPerMenuRow* parameter. The number of graphic text rows that are dedicated to the layer menu viewport is defined by the value of the *kpNumLayerMenuRows* parameter. If the size of the coarse window is sufficiently large such that the fine window is to be displayed in the fine viewport, the *kpDisplayFineViewport* parameter is set to the logical value of *True;* otherwise, if only the coarse window is displayed in the large coarse viewport, the parameter *kpDisplayFineViewport* is set to the value of *False*. The *kpRedisplayControl* member of the parameters structure controls the effect of a display routine in the layout viewport. The value of *kpRedisplayControl* specifies the current mode of display and can be one of three options that are defined as follows in the *kic.h* file:

```
/*
 * Viewport control flags
 */
#define   SPLITSCREEN             'b'
#define   FINEVIEWPORTONLY        'f'
#define   COARSEVIEWPORTONLY      'c'
```

If both the fine and small coarse viewports are displayed, the *kpRedisplayControl* parameter will be assigned the value of SPLITSCREEN by default, and the *kpDisplayFineViewport* parameter will be set to the value of *True*. If only the large coarse viewport is displayed, then *kpRedisplayControl* will assume the value of COARSEVIEWPORTONLY by default, and the *kpDisplayFineViewport* parameter will be set to the value of *False*. By setting the display control parameter to the value of FINEVIEWPORTONLY, only the fine viewport will be updated by a display routine, and the COARSEVIEWPORTONLY switch will result in only the small or large coarse viewport to be effected by a display routine. Any procedure that uses the display control parameter to control the KIC geometry display routines must reset the parameter to it's default value before terminating.

The window stack of the current symbol is represented by the *kpWindowStack* and *kpNumWindows* members of the parameters structure. See Section 4.2.2 that describes the window stack descriptor.

A user interrupt condition is identified by the *kpSIGINTERRUPT* member of the parameters structure. Whenever the KIC user presses the interrupt key (break or delete under UNIX and control C for VMS), the *kpSIGINTERRUPT* parameter is set to the value of *True* by the *CatchSIGINT()* interrupt handling procedure. The top of the display loop (i.e., the top of the CD generator loop before the *CDGen()* procedure is invoked to return a pointer to an object in the window to be displayed) the value of the *kpSIGINTERRUPT* is tested for an interrupt condition. If an interrupt had occurred, the display routine is terminated and the *kpSIGINTERRUPT* parameter is returned to its default value of *False*.

### 4.4. KIC Command Menus

Because of the functionality of KIC, it is impossible to display all KIC commands simultaneously in a menu structure. KIC therefore is designed with a hierarchical menu system that provides specially tailored command sets allowing the user to easily complete specific tasks. These command sets are not so specific or restrictive to frequently burdened the user with the necessity to traverse the command hierarchy for the desired commands.

The present implementation of the KIC command menu system is slightly awkward. This method has continued in use simply because it works and has not yet become inconvenient. This section explains the command menu operation, and provides a description of the task of adding a new menu command. Also, suggestions for improving the KIC command menu system are provided.

There are seven possible hierarchical command menus for KIC: the basic menu at the top of the command hierarchy, the selection menu for object modification, the instance menu for instance placement, the attribute menu for defining the viewport display attributes, the property menu for editing the property lists of selected objects, the ambiguity menu for resolving ambiguities in instance selection, and the debug menu that the normal KIC user should never use. The command menu that is currently displayed in the command menu viewport is identified by the *kpMenu* member of the KIC parameters structure that can assume one of the values defined in the *kic.h* file as follows:

```
/*
 * Menu names
 */
#define   BASICMENU          'b'
#define   DEBUGMENU          'd'
#define   SELECTIONMENU      's'
#define   INSTANCEMENU       'i'
#define   ATTRIBUTESMENU     'a'
#define   PROPERTYMENU       'p'
#define   AMBIGUITYMENU      'A'
```

A command menu in KIC is an array of character string pointers, each pointer referencing a particular command name. For example, the basic menu is defined in the *kic.h* file as follows:

```
/*
 * KIC menus.
 */
#ifdef Allocate
          char *BasicMenu[] = {
                " ",
                "EDit",
                "DIR",
                "SAve",
                "WRite",
                " ",
                "ATtri",
                "Insta",
                "SElec",
                "PRpty",
                " ",
                "RDraw",
                "EXpnd",
                "PEek",
                "PAn",
                "Zoom",
                "WINdo",
                "VIEw",
                "LASt",
                " ",
                "45s",
                "Grid",
                "SNap",
                "BOXes",
                "WIRes",
                "WIDth",
                "POLyg",
                "DOnut",
                "FLASh",
                "ARC",
                "LABel",
                "Undo",
                " ",
                "LYra",
                "TECh",
                "ABort",
                "DEBug"
                };
          int NumBasicMenu = sizeof(BasicMenu)/sizeof(char *);
#else
          char *BasicMenu[];
          int NumBasicMenu;
#endif
```

There must be such an array of character pointers for each command menu in KIC.

Notice that the contents of the command menu are defined by using of a compile flag named *Allocate*. This compile flag is set to a non-zero logical value by one and only one source file that includes the *kic.h* header file; typically it is defined in the source file that contains the *main()* procedure.

Also notice that the command names are of mixed case with the capitalized letters always forming a unique prefix. The convention has become that the first, capital letters in the command name specify the minimum number of characters required to identify the particular command name from any other KIC command; *this convention must be obeyed* because the *Point()* routine, which compares the user keyboard input to the current command set and thereby allows commands to be selected through the keyboard, will compare the user keyboard input to only those characters in the command names that are capitalized.

The major problem with the KIC command menu structure is that the menus are defined entirely in the *kic.h* header file. As a result, it is necessary to completely recompile KIC whenever a new command is added to a menu or the whenever the structure is modified. The preferred solution to the menu problem might be to have a specific source file, say, menu.c, containing the routines for managing the command menu viewport.

The *ShowMenu()* procedure will display a particular command menu in the command menu viewport. The *MenuSelect()* procedure command is invoked to highlight a specific command menu item whenever the command is user-selected, and the *MenuDeselect()* procedure is invoked to return a specific command menu item to its default, unhighlighted appearance. There is typically a menu-specific display procedure for each command menu, such as the

*ShowBasicMenu()* procedure, that in turn invokes the *ShowMenu()* routine. The reason for this one level of indirection is that typically several command items in a command menu will reflect a mode of operation that is identified by a particular member of the parameters structure; for example, the expand command that will cause all called symbols to be displayed in full detail in all layout viewports. The menu-specific command menu display procedure will be responsible for testing the various modes of KIC and highlighting the identifying command menu item if the respective mode is in effect.

The *kic.h* header file also has specific character pointers defined for each possible command menu name for reasons of comparison. Several such pointers for the commands in the basic menu are defined as follows in the *kic.h* file:

```
#ifdef Allocate
            char *MenuEDIT  = "EDit";
            char *MenuDIR   = "DIR";
            char *MenuSAVE  = "SAve";
            char *MenuWRITE = "WRite";
                    .
                  etc.
                    .
#else
            char *MenuEDIT;
            char *MenuDIR;
            char *MenuSAVE;
            char *MenuWRITE;
                    .
                  etc.
                    .
#endif
```

These character pointers are typically used to determine whether a command menu item has been user-selected. As is described in Section 4.3 on the KIC parameters structure, when a command in the current command set is selected, whether by use of the graphical pointing device or keyboard input, the name of the command as it appears in the command menu viewport is placed into the *kpCommand* character buffer in the KIC parameters structure by the *Point()* routine. The procedure that monitors the command menu would therefore test the returned value of *kpCommand* for the appearance of a new command name. For example, the following sections of C code could be extracted from the *Basic()* routine:

```
#include "kic.h"
Basic(){
     int LookedAhead = False;
       .
       .
       .
     loop {
          if(LookedAhead)
               LookedAhead = False;
          else
               Point();
          if(strcmp(Parameters.kpCommand,MenuEDIT) == 0){
               .
               etc.
               .
               }
          else if(strcmp(Parameters.kpCommand,MenuABORT) == 0){
               .
               etc.
               .
               }
          else if(strcmp(Parameters.kpCommand,MenuEXPND) == 0){
               .
               etc.
               .
               }
            .
            etc.
            .
          }
     }
```

In the *Basic()* routine, the program will loop continuously until the user invokes a command to break the loop such as the abort command. This is typical of all command menu routines in KIC.

Because the KIC command menus are hierarchical, the *Basic()* routine will invoke procedures such as *Instances()*, *Sel()*, *Attri()*, *Debug()*, and *Properties()* that will erase the current command viewport, display a specific command menu, and begin looping with the pointing device as is done in the *Basic()* routine. The only argument that is passed to these menu routines is a pointer to the integer *LookedAhead* that is used by the subroutine to notify the parent routine that the user has already selected a menu command. When the logical value of the *LookedAhead* integer is nonzero, the *Point()* routine is not invoked during that particular pass through the loop.

### 4.5. The Pointing Device

The graphical pointing device is extremely important to KIC because it is used as the major communication link between the editor and the KIC user. Consequently, considerable thought has been given to its application for layout entry and viewport control. This section will survey the important *Point()* routine in KIC.

KIC demands that the graphical pointing device be the locator type; a user pointing event would return to the host the display coordinate that was user-specified and a mask that would identify the button that was pushed on the pointing device or keyboard. The *Point()* routine will map the display coordinate to the appropriate viewport and, on the basis of the position and the button mask, will attempt to determine the intent of the KIC user.

If the graphical pointing device does not have special buttons, such as the commonly used four button mouse has, KIC will assume that the user is pointing to a lambda coordinate in the layout viewport if the returned display coordinate is within the layout viewport and the if the user pressed the space bar on the keyboard. This being the case, the KIC user would use the space bar of the terminal whenever he pointed to a lambda coordinate in the layout viewports.

The *Point()* is responsible for acting on the occurrence of several keyboard-specific commands. The following table lists the special keyboard commands of KIC and describes the actions that are performed when the event occurs:

control-A    Control A will cause KIC to abort unconditionally. This keyboard command could be dangerous, and therefore it is possible to remove it from the *Point()* routine by recompiling without the ABORT compiler flag in the point.c file.

control-C    This keyboard command is available if and only if KIC is compiled to run under UNIX. After the user types control-C, KIC will prompt the user for a lambda coordinate that will be accepted as if the user had pointed to that lambda coordinate in the layout viewport. The *Point()* routine will return with the *kpPointCoarseViewport* member of the KIC parameters structure set to the logical value of *True*.

control-E

This keyboard command is identical to the above control-C command. It is intended for VMS systems for which the control-C command produces the terminal interrupt signal SIGINT.

control-F

After typing control-F, KIC will prompt the user to identify a new center of the fine window in the current symbol. After the user has specified the new center, the fine window will be redrawn in its new position. The relative size of the fine window will not be affected by this operation. Also, the *Point()* routine will not terminate after this user command; as a result, the control-F keyboard command is transparent to whatever procedure invokes *Point()*.

control-G

When the user types control-G, KIC will prompt the user to identify a new size and position for the fine window in the current symbol. After the user has specified this information by pointing to the endpoints of the diagonal of the new fine window, the fine window will be redrawn in its new position. Because the fine viewport has a fixed aspect ratio, the horizontal width of the new fine window that the user specifies will take precedence. This keyboard command is similar to the control-F command in that the *Point()* routine will not return after this user command; as a result, the control-G keyboard command is transparent to whatever procedure invokes *Point()*.

control-L

When the user types control-L, KIC will prompt the user for a positive integer that will identify the new current layer. The number one identifies the first mask layer in the KIC layer table, etc. The *Point()* routine will return with the *kpPointLayerTable* member of the KIC parameters set to the value of *True* whenever the control-L keyboard command is used.

control-N

This keyboard command adds another item to the window stack of the current symbol. After typing control-N, KIC will prompt the user to name the current coarse and fine windows. When the KIC user has assigned a name to the current layout windows, the windows are pushed onto the *kpWindowStack* window stack member of the KIC parameters structure, and the value of *kpNumWindows* is incremented by one. This keyboard command is similar to the control-F command in that the *Point()* routine will not terminate after this user command.

control-T

This keyboard command changes the size and aspect ratio of the fine window and viewport. If the fine viewport occupies the bottom third of the layout area of the graphics display and the KIC user types control-T, the layout viewports be recomputed and redrawn such that the fine window will occupy the left half of the layout area of the display. The new fine window will have the same width in lambda as the previous fine window and will have the same center position. This keyboard command is similar to the control-F command in that the *Point()* routine will not terminate after the command.

control-V

This keyboard command is identical to the above control-T command. It is intended for VMS systems for which the control-T command has has a special meaning to the system.

control-W

This keyboard command provides the option of a '*where am I*?' command. When the user types control-W, KIC will then expect the user to point in the layout viewports, and will notify the KIC user of the lambda coordinate value by invoking the *ShowXY()* procedure. This keyboard command is similar to the control-F command in that the *Point()* routine will not terminate after the command.

character

Whenever the KIC user types a printable character on the keyboard while the graphical pointing device is active, the character is received by the *Point()* routine, converted to lower case, and placed at the end of the contents of the *kpCommand* buffer in the KIC parameters structure. The *Point()* procedure maintains a running count of the number of buffered keyboard characters and will not immediately terminate after the user types a character. After the typed character has been buffered, the contents of the *kpCommand* buffer are compared with the current command set that is displayed in the command menu viewport. If the *Point()* procedure determines that the user has typed a command name, the procedure returns with the complete command name in the *kpCommand* buffer.

escape

When the KIC user presses the escape button on the keyboard, all buffered keyboard characters are cleared. The *Point()* procedure does not return after the user presses the escape key.

exclamation (!)

The exclamation point is the only printable character that has special meaning to the *Point()* procedure. The exclamation point is used as the KIC system interface; when the KIC user types an exclamation point, KIC will expect the user to type a system command terminated by a carriage return. The command string will then be passed to the *ShowProcess()* routine that will execute the command and display any output in the area of the fine viewport. When the user presses a key on the keyboard to signify that he has read the displayed output, the fine window is redrawn. The *Point()* procedure does not return after the user executes a process in the fine viewport.

If the KIC user points in the layer menu viewport with the graphical pointing device, the *PointLayerTable()* routine is invoked to determine if the user actually pointed at a valid layer. If the user user did select a new current layer, the *kpLayer* member of the KIC parameters structure is set to the index of the new current layer in the KIC layer table, and the *Point()* procedure returns with the *kpPointLayerTable* member of the KIC parameters structure set to the logical value of *True*. The *kpPointLayerTable* is typically used by procedures such as *Fille()*, *Blink()*, or *Visib()* that require the user to point in the layer menu viewport to identify layers to be assigned specific attributes. As described above, the KIC user can also select a new current layer with the control-L keyboard command.

When the user points in any layout viewport with the graphical pointing device, the viewport coordinate is converted to the respective window coordinate by the *CtrlAt()* procedure that is invoked by the *Point()* procedure, and the values are placed in the *kcX*, *kcY* members of the KIC cursor descriptor described in Section 4.2.4. The previous values of *kcX*, *kcY* are moved to *kcPredX*, *kcPredY*, and the displacement is computed and placed in the *kcDX*, *kcDY* members of the KIC cursor descriptor. The *CtrlAt()* procedure also sets the *kpPointCoarseViewport* member of the KIC parameters structure to the value of *True* and invokes the *ShowXY()* routine to display the current cursor information in the information viewport. The geometry input procedures, such as *Boxes()*, *Wires()*, and *Polygons()* will wait for the *kpPointCoarseViewport* flag to indicate that there is new information in the cursor descriptor. The *Point()* procedure will clear the *kpCommand* buffer in the KIC parameters structure and return immediately after invoking the *CtrlAt()* procedure.

To repeat from the above table of keyboard commands, when the KIC user types a printable character on the keyboard while the graphical pointing device is active, the character is received by the *Point()* routine, converted to lower case, and placed at the end of the contents of the *kpCommand* buffer in the KIC parameters structure. The *Point()* procedure maintains a running count of the number of buffered keyboard characters and will not immediately terminate after the user types a character. After the typed character has been buffered, the contents of the *kpCommand* buffer are compared with the current command set; the current command set is identified by the *kpMenu* member of the KIC parameters structure. If the *Point()* procedure determines that the user has typed a command name, the procedure returns with the complete command name in the *kpCommand* buffer.

KIC will be easiest to use if the graphical pointing device has at least four buttons. If the graphical pointing device has buttons, the *fButtons* member of the frame buffer descriptor is set to the value of *True*, and the number of buttons on the graphical pointing device is specified by the *fNumButtons* member of the frame buffer descriptor that is described in Section 4.6.1. When the user pushes one of these buttons on the pointing device, the terminal will report a button mask to the host, and this button mask will be compared with the array of button masks referenced by the *fButtonMask* member of the frame buffer descriptor to identify the respective button that was pushed. The first integer in the *fButtonMask* array is the value of the mask for the first button on the pointing device, etc.

In KIC, the first button on the graphical pointing device is used specifically for identifying entries in the command or layer menus and for specifying a window coordinate whenever KIC is prompting the user for one. The second pointing device button is used for repositioning the center of the fine window, and the fourth button is used to redefine the size of the fine window from two user-specified window coordinates of the new fine window diagonal. Because the size of the fine viewport is fixed, the size of the new user-specified fine window will be clipped to fit into the fixed aspect ratio. The third button on the graphical pointing device performs the same function as the control-W keyboard command; when the points in the layout viewports by pressing the third pointing device button, the respective window coordinate is displayed in the information viewport by the *ShowXY()* procedure. The third pointing device button is typically used for measuring the size of objects in the database. The *Point()* procedure will terminate after the KIC user presses the second, third, or fourth pointing device button. Only the first pointing device button will cause the *Point()* procedure to return with a new window coordinate in the KIC cursor descriptor.

### 4.6. The Frame Buffer Interface

As presented in Chapter 1, KIC is designed to run on a wide range of raster graphics terminals or frame buffers and indeed runs on several devices including the AED 512 and 767, the Tektronix 4113 and 4105, the HP2648A, the Metheus Omega-400, and the Masscomp MC500. The frame buffer is modeled by the contents of the KIC frame buffer descriptor that contains the boolean and numeric capabilities describing the respective frame buffer characteristics. This report assumes that the reader is familiar with the basic structure of a raster color graphics terminal, including the use of video memory and the color look-up table. For more information on this subject, see [6] and [7].

#### 4.6.1. The Frame Buffer Descriptor

The frame buffer descriptor is defined as follows in the *fb.h* header file.

```
/*
 * Frame Buffer desc.
 */
struct f {
     char *fDisplay;
     char *fDeviceName;
     int *fButtonMask;             /* pointer to array of button masks */
     int fMaxX,fMaxY;              /* raster dimensions of viewport */
     int fFontHeight;              /* standard font size */
     int fFontWidth;
     int fMaxIntensity;            /* set to 255 (normalized intensity) */
     int fMaxP;                    /* max pixel intensity */
     int fNumColors;               /* max color index */
     int fNumRows;                 /* max number of horizontal rows */
     int fNumColumns;              /* max number of vertical columns */
     int fNumFillPatterns;         /* max fill index */
     int fInitialized;             /* FB struct is initialized */
     int fNonDestructiveText;      /* text doesn't wipe background */
     int fLastCursorColumn;        /* last column used by ShowPrompt() */
     int fFilledPolygons;          /* frame buffer has filled polygons */
     int fDefinableFillPatterns;   /* fill styles are definable */
     int fButtons;                 /* pointing device has buttons */
     int fNumButtons;              /* number of pointing device buttons */
     }
     FB;
```

The *fDisplay* member of the frame buffer structure is a pointer to a character string that specifies the name or type of the respective graphics terminal;

for example, the Tektronix 4113 might be named "t5". The *fDeviceName* is a pointer to the name of the system device driver for the graphics device. An example of a UNIX device name might be "*/dev/tty01*".

The resolution of the graphics display is defined by the *fMaxX* and *fMaxY* frame buffer structure members. If the display size of the graphics device is characterized by 640 horizontal pixels and 480 vertical pixels, the values of *fMaxX* and *fMaxY* would be 639 and 479 respectively.

The maximum color index for the frame buffer is defined by the *fNumColors* member of the frame buffer descriptor. For a monochromatic display, this structure member would be assigned the value of unity. Zero is a valid color index and is always used by KIC as the background color. For a color display, the maximum gun intensity for red, green, or blue is defined by the value of the *fMaxP* structure member. The *fMaxIntensity* structure member is the value to which KIC normalizes the gun intensities and should always be set to the value of 255. In other words, KIC assumes that the gun intensity for the color display is an eight bit value. The *FBVLT()* routine, which redefines an entry in the frame buffer's color look-up table, will convert a color intensity of 255 to the value of the *fMaxP* structure member.

The maximum index of fill patterns for the frame buffer is given by the value of the *fNumFillPatterns* frame buffer structure member. Zero is assumed to be the index for a solid-fill pattern. If the fill patterns are definable, then the *fDefinableFillPatterns* structure member is set to the value of *True*, and if the frame buffer is capable of filled polygons, including solid fill, then the *fFilledPolygons* structure member is also set to the value of *True*. The polygon display mechanism for graphics devices that are not capable of filled polygons is to draw a line around the perimeter of the object.

The pixel size of the font array for graphic text is defined by the *fFontWidth* and *fFontHeight* members of the frame buffer structure. From these two structure members and the *fMaxX* and *fMaxY* structure members, the values of *fNumRows* and *fNumColumns* are computed; *fNumRows* specifies the number of graphic test rows from the top to bottom of the graphics display, and *fNumColumns* contains the number of graphic text columns from the left to right of the display. The size of the graphic text font should be sufficiently small to allow the value of the *fNumRows* structure member to be greater than 30. See Section 4.1.1 that describes the graphic text coordinate system in menu and information viewports.

When KIC invokes the *ShowPrompt()* routine to display text in the prompt viewport, the last textual column in which text is displayed is saved in the *fLastCursorColumn* frame buffer structure member. The *FBKeyboard()* routine, which obtains a user-typed character string from the keyboard of the graphics terminal, will use this information to decide where to begin to echo user-typed text in the prompt viewport.

If the graphic text is not destructive, i.e., the background in the character font array is not changed when a graphic character is displayed, the *fNonDestructiveText* frame buffer structure member is set to the value of *True*. It is preferable for graphic text to be non-destructive. If the text is destructive, the display of layout information may be corrupted by a textual label, and KIC will also highlight command menu items differently in the *MenuSelect()* procedure.

It is assumed that the frame buffer has some form of graphical pointing device. If the graphical pointing device has buttons that send locator reports to the host such that the host can determine which button was pressed by the KIC user, the *fButtons* structure member is set to the value of *True*. The number of buttons on the pointing device is given by the *fNumButtons* structure member. The *fButtonMasks* pointer references an array of integers that are used to identify the buttons of the graphical pointing device. If the KIC user presses the first button on the pointing device, the *FBPoint()* routine will return with the respective button mask that was received by the host from the frame buffer. The *Point()* procedure will recognize that the first button was pushed because the button mask value will be identical to the first integer in the array referenced by the *fButtonMasks* pointer, etc. KIC uses no more than four buttons on the graphical pointing device.

The *fInitialized* structure member is set when the frame buffer descriptor is completely initialized and the graphics device driver is in CBREAK mode; see *tty(4)* in the BSD UNIX programmer's manual. CBREAK mode implies that characters that are typed at the terminal are not buffered, but becomes available to KIC when they are typed. Also, character·echo must be disabled; KIC will echo the keyboard input. Flow control and interrupt processing should always be enabled.

### 4.6.2. The Frame Buffer Routines

The frame buffer routines that are required for KIC to control a graphics device or frame buffer are described in this section. There should be sufficient information provided here for any programmer to write a display driver for KIC. However, the best procedure for interfacing KIC to a new graphics terminal is to take an existing set of frame buffer routines for another graphics terminal and modify the routines for the new graphics device.

All graphics device dependencies are managed by these frame buffer routines, and, as a result, a set of frame buffer routines that were written for a particular frame buffer will probably differ greatly from routines that were written for another graphics device. Separate display drivers have been written for several graphics terminals and graphics work stations as well as the *Model Frame Buffer* terminal independent graphics package.

The following is a synopsis of all frame buffer routines that are required by KIC for any graphics device:

```
FBBegin(Display)
char *Display;

FBInitialize()

FBHalt()

FBEnd()

FBMoveTo(X1,Y1)
int X1,Y1;

FBDrawLineTo(X2,Y2)
int X2,Y2;

FBLine(X1,Y1,X2,Y2)
int X1,Y1,X2,Y2;

FBForeground(DisplayOrErase,ColorId)
char DisplayOrErase;
int ColorId;

FBBox(ColorId,DisplayOrErase,Type,StyleId,Left,Bottom,Right,Top)
char Type,DisplayOrErase;
int StyleId,ColorId,Left,Bottom,Right,Top;

FBDefineFillPattern(StyleId,BitArray)
int StyleId;
int *BitArray;

FBSetFillPattern(StyleId)
int StyleId;

FBVLT(ColorId,Red,Green,Blue)
int ColorId,Red,Green,Blue;

FBText(Mode,RowOrX,ColumnOrY,Text)
int RowOrX,ColumnOrY;
char *Text;
char Mode;

FBPolygon(ColorId,Type,StyleId,xy,ncoords)
int *xy;
int ColorId,StyleId,ncoords;
char Type;

FBBlink(ColorId,Red,Green,Blue,Flag)
int ColorId, Flag;
int Red,Green,Blue;

FBFlood()

FBSetCursorColor(ColorId)
int ColorId;

FBPolygonClip(xy,ncoords,window)
int *xy,*ncoords;
struct ka window;
```

```
FBTransfer()

FBKeyboard(TypeIn)
char **TypeIn;

FBPoint(X,Y,Key,Buttons)
int *X,*Y,*Buttons;
char *Key;

FBMore(Left,Bottom,Right,Top,Textfile)
int Left,Bottom,Right,Top;
FILE *Textfile;
```

KIC assumes that all frame buffer routines work properly and therefore are not required to return a diagnostic value.

The *FBBegin()* routine is the first frame buffer routine invoked by KIC and initializes the frame buffer descriptor described above for the respective graphics device. The only argument *Display* specifies the display name or type and becomes the *fDisplay* member of the frame buffer descriptor. The *fInitialized* member of the frame buffer descriptor is used to define the state of the graphics device driver and is typically not set by the *FBBegin()* procedure.

The *FBBegin()* routine will invoke the *FBInitialize()* procedure to initialize the graphics device driver to the required state. If the *fInitialized* member of the frame buffer descriptor is set to zero, the *FBInitialize()* routine will save the current state of the graphics device driver and then set the state of the driver to a CBREAK mode with no character echoing; see *tty(4)* in the BSD UNIX programmer's manual. CBREAK mode allows a character to become available to the user program as soon as it appears from the terminal and also permits flow control. If the *fInitialized* structure member is not set, then typically there is no action when *FBInitialize()* is invoked. When the communication link with the graphics device is not over *stdio*, such as for a Metheus Omega 400 display controller which does not have a keyboard, it is also desirable to place the standard input in CBREAK mode; when KIC requests the user to input through the keyboard, the user input will be taken from the standard input device and echoed normally on the graphics display. The *fInitialized* member of the frame buffer descriptor is set to the value of *True* before the *FBInitialize()* procedure terminates.

There are two procedures for releasing control of the graphics display or frame buffer: *FBHalt()* and *FBEnd()*. The *FBHalt()* routine is invoked when KIC receives a SIGTSTP stop signal (UNIX only) which is generated by the KIC user from the keyboard when he wants to suspend, but not terminate, the KIC program. The *FBEnd()* routine is however the last frame buffer procedure invoked by KIC. Both procedures will clear the graphics display, return the device driver to its original state, and set the *fInitialized* member of the frame buffer descriptor to the value of *False*. If the KIC program was suspended by the user, invocation of the *FBInitialize()* procedure will again save the current state of the device driver and set the driver to the required CBREAK mode.

A common configuration for running KIC is to have a graphics terminal connected to the host over a serial, full duplex, RS-232 line with a 9600 or 19200 Baud rate. When this is the case, it is always necessary to use buffered output to reduce the number of system I/O requests to the host computer. An example C procedure for buffering output characters is shown below:

```
#define   BUFSIZE  4096

static    int      NumTTYBuffer = 0;
static    char     TTYBuffer[BUFSIZE];

FBWrite(cp,n)
     char *cp;     /* pointer to string to be put in buffer */
     int n;        /* size of string to be put in buffer */
     {
     /* test for buffer overflow */
     if((NumTTYBuffer + n) >= BUFSIZE){
          write(1,TTYBuffer,NumTTYBuffer);
          NumTTYBuffer = 0;
          }
     while(n--)
          TTYBuffer[NumTTYBuffer++] = *cp++;
     }
```

If the output graphics code is buffered, it is frequently necessary to flush the output buffer or transfer all buffered characters to insure that a specific display action will occur immediately. The *FBTransfer()* procedure would accomplish this, as in the above C procedure, by writing all buffered characters to the graphics device and setting the buffered character count to zero. If output is not buffered, the *FBTransfer()* routine will perform no operation.

The *FBSetFillPattern()* routine is invoked to set the current fill pattern of the frame buffer to that identified by the integer *StyleId* that is greater than or equal to zero and less than or equal to the value of *fNumFillPatterns* in the frame buffer descriptor. Solid fill is always defined by *StyleId* equal to zero. If the graphics device does not provide filled geometries or has only solid fill, the *FBSetFillPattern()* procedure will perform no operation.

The *FBDefineFillPattern()* procedure defines the fill pattern identified by *StyleId* and returns with *StyleId* as the current fill style. As for the *FBSetFillPattern()* procedure, the value of *StyleId* is greater than or equal to zero and less than or equal to the value of *fNumFillPatterns* in the frame buffer descriptor. The *BitArray* argument is a pointer to an array of eight integers whose least significant eight bits represent individual rows in an eight by eight intensity array. For example, a fill pattern with an ascending diagonal line may be defined by the following eight (decimal) integers:

1 2 4 8 16 32 64 128 256

A diagonal-grid fill pattern can be defined with the following integer array:

257 130 68 40 40 68 130 257

The value of the *ColorId* argument for all frame buffer routines must be greater than or equal to zero and less than or equal to the value of *fNumColors* in the frame buffer descriptor. The *ColorId* argument is in fact the index of a particular display color in the video color table of the respective graphics device, and the *fNumColors* member of the frame buffer descriptor specifies the size of the color table.

The color map that is used by KIC is simple; the first color in the video color table, which corresponds to a *ColorId* value of zero, is always the background color of the display. The last color in the video color table, which corresponds to a *ColorId* value equal to the *fNumColors* member of the frame buffer descriptor, is always used for highlighting objects in the selection queue, displaying the coordinate axes, and also defines the color of the cursor. The remaining colors are dedicated to the display of mask geometries. Two mask layers can not share the same color index of the video color table unless there are insufficient display colors in which case the several mask layers will share the last entry in the color table, the highlighting color.

The *FBVLT()* procedure defines the video color table entry for the color identified by *StyleId* to be the color represented by the *Red*, *Green*, *Blue* combination. The values of *Red*, *Green*, and *Blue* are normalized to the value of the *fMaxIntensity* member of the frame buffer descriptor, which is usually equal to 255, and their absolute maximum value is specified by the *fMaxP* frame buffer structure member. Once the color corresponding to *ColorId* is redefined, all geometries that were written into the display memory of the respective frame buffer will immediately be displayed in the new color. The value of the *ColorId* argument must be greater than or equal to zero and less than or equal to the value of *fNumColors* in the frame buffer descriptor.

The *FBForeground()* routine is invoked to set the current drawing color to that specified by the *ColorId* argument. The *DisplayOrErase* argument specifies whether the color is to be displayed or erased, and it must be one of the values defined as follows in the *fb.h* header file:

```
#define   ERASE          'e'    /* Erase to background color */
#define   DISPLAY        'd'    /* Show object as specified */
```

If the color is to be erased, it is assumed that it will be erased to the background color. The *DisplayOrErase* argument is redundant if the foreground color is set to color zero, the background color. It is provided for the programmer who wishes to write a frame buffer driver that uses a special *ALU mode* which requires this information.

To explain the meaning of the ALU mode, it is necessary to understand the procedure by which a geometry is written into the video memory of the graphics display. If we assume that the display is a typical raster device, each pixel of the display is represented by a location in the video memory, and that memory location contains the index in the video color table for the color by which the respective pixel is to be displayed on the CRT. Most frame buffers provide the capability of specifying how video display memory is changed by a write operation. One common mode of writing into the video memory is an arithmetic 'OR' operation between the source color index and the contents of the destination memory location, where the source color index is the current, foreground color index. In this case, the ALU mode is the 'OR' mode. KIC will in general use the 'replace' mode in which the contents of the video display memory are replaced by the source color index, regardless of the previous contents of the display memory.

The *FBFlood()* procedure is invoked to clear the entire display to the current, foreground color.

The *FBSetCursorColor()* routine is invoked to set the color of the graphical cursor to the color identified by *ColorId*. As described above, the last entry in the video color table of the frame buffer is always used for the cursor color.

KIC allows mask colors to blink, if the respective frame buffer has this capability. If the *FBBlink()* routine is invoked with the *Flag* argument equal to the value of *True*, the color that is identified in the video color table of the graphics display by the *ColorId* index is set to blink between its normal color and the color defined by the color combination of the arguments *Red*, *Green*, and *Blue*. The normal color for any given value of *ColorId* is the color that is defined in the video color table. The rate of blinking is not definable; colors should blink at a rate that is comfortable to view. If the *FBBlink()* routine is invoked with the value of the *Flag* argument set to *False*, then the color identified by the value of *ColorId* is set to a non-blinking state. If the respective frame buffer does not provide the capability of blinking colors, or alternating color definitions in the video color table, the *FBBlink()* procedure performs no operation.

There are six routines for displaying figures or geometries on the graphics display: *FBMoveTo()*, *FBDrawLineTo()*, *FBLine()*, *FBBox()*, *FBPolygon()*, and *FBText()*. These frame buffer routines will display an object or figure in the current, foreground color and with the current fill pattern, if applicable. All coordinates that are passed as arguments to these frame buffer procedures are display coordinates with there *(0,0)* origin in the bottom, left corner of the graphics display. The effect of these routines might be delayed until the *FBTransfer()* procedure is invoked to flush the output.

The *FBMoveTo()* procedure sets the current graphics position to the display coordinate identified by the *X1,Y1* arguments. A subsequent invocation of the *FBDrawLineTo()* procedure will produce a solid line from the current graphics position to the display coordinate identified by the *X2,Y2* arguments, and the current graphics position then moves to the latter display coordinate. The *FBLine()* routine is invoked to display a solid line from the *X1,Y1* display coordinate to the *X2,Y2* display coordinate, after which the current graphics position is set to the *X2,Y2* display coordinate.

The *FBBox()* routine is invoked to display or erase a box in the color identified by the *ColorId* argument and in the pattern identified by the *Type* and

*StyleId* arguments. The resulting box is defined by the display coordinates *Left*, *Bottom*, *Right*, *Top* and is either displayed in the color identified by *ColorId*, which becomes the foreground color, or is erased to the background color depending on the value of the *DisplayOrErase* argument. The *DisplayOrErase* argument must be set to one of the values defined as follows in the *fb.h* header file:

```
#define   ERASE          'e'   /* Erase to background color */
#define   DISPLAY        'd'   /* Shoe object as specified */
```

The *Type* argument specifies whether the box is to be filled or outlined, and this argument must be one of the values defined as follows in the *fb.h* header file:

```
#define   FILL           'f'   /* Fill with current pattern */
#define   OUTLINE        'o'   /* Outline contour of object */
```

If the box is filled, the *StyleId* argument defines the fill pattern to be used; this fill style becomes the current fill pattern. The box must always be filled whenever the *DisplayOrErase* argument specifies an erase operation.

*FBPolygon()* is invoked to display a polygon in the color identified by the *ColorId* argument and in the pattern identified by the *Type* and *StyleId* arguments. The vertices of the polygon are defined in the integer array referenced by the *xy* pointer; the number of vertices is specified by the *ncoords* argument, and the first display coordinate is *(xy[0], xy[1])*, the second display coordinate is *(xy[2], xy[3])*, etc. The *Type* argument specifies whether the polygon is to be filled or outlined, and this argument must be one of the values defined as follows in the *fb.h* header file:

```
#define   FILL           'f'   /* Fill with current pattern */
#define   OUTLINE        'o'   /* Outline contour of object */
```

If the polygon is filled, the *StyleId* argument defines the fill pattern to be used; this fill style becomes the current fill pattern. If the frame buffer is not capable of displaying filled polygons, the polygons are always outlined.

*FBText()* is invoked to display a character string referenced by the *Text* argument at the position defined by the *RowOrX* and *ColumnOrY* arguments. As described in Section 4.1.1, KIC uses two coordinate systems for the positioning of graphic text, and the *FBText()* procedure must accommodate both systems. The *Mode* argument specifies which coordinate system is to be used and must be set to one of the values defined as follows in the *fb.h* header file:

```
#define   ROW_COLUMN          'r'   /* use ASCII terminal mode */
#define   PIXEL_COORDINATE    'p'   /* use graphics display mode */
```

If the *Mode* argument is set to the value of ROW_COLUMN, the *RowOrX* and *ColumnOrY* arguments specify the position for the first character of the string in the row-column or character block coordinate system; otherwise, these arguments specify the lower, left display coordinate for the first character of the string. Displayed text is never rotated and normally uses a single character font. Because the row-column coordinate system is used exclusively for displaying text in the command, information, and layer menu viewports, the *Mode* argument could be used, however, to specify one of two graphic text fonts.

*FBPoint()* is the only frame buffer routine that controls the graphical pointing device. When invoked, the *FBPoint()* routine displays the graphical cursor and then waits for a user pointing event. If the graphical pointing device is equipped with buttons, a pointing event occurs when the KIC user presses one of these buttons or presses a character key on the keyboard; otherwise, if there are no buttons on the graphical pointing device, a pointing event occurs only when the user presses a character key on the keyboard. After one user pointing event, the routine terminates, and the returned values also depend on whether the graphical pointing device is equipped with buttons. If the graphical pointing device has buttons and the user pressed one of these buttons, *FBPoint()* returns with the value of zero in the character referenced by the *Key* pointer, the display coordinate coordinate of the graphics cursor (i.e., the position of the cursor) in the integers referenced by the *X*, *Y* pointers, and the identifying button mask in the integer referenced by the *Buttons* pointer; this button mask can be compared with the contents of the *fButtonMask* member of the frame buffer descriptor to determine the button that was pressed. If the graphical pointing device has buttons and the KIC user types at the keyboard of the graphics terminal, then the character that was user-typed is returned in the character referenced by the *Key* pointer, and the other returned values are meaningless. If the graphical pointing device does not have buttons and the user presses a character key on the keyboard, the *FBPoint()* routine returns with the value of typed character in the character referenced by the *Key* pointer, the display coordinate of the cursor position in the integers referenced by the *X*, *Y* pointers, and the returned value of the *Buttons* argument is meaningless. The graphics cursor is always disabled before *FBPoint()* terminates.

The *FBKeyboard()* procedure is invoked to obtain user input from the device keyboard or the standard input. This input routine will perform character buffering and input line management, and return the pointer *TypeIn* to the input character buffer. The input line management includes the handling of the erase character (typically control-H or delete), the line kill character (control-X or control-U), special hacking of control characters (especially the escape character), and character echoing. All characters that are typed by the KIC user are echoed in the information viewport beginning at the text column specified by the *fLastCursorColumn* member of the frame buffer descriptor, and are displayed with the color identified by the *kpMenuTextColor* member of the KIC parameters structure, which becomes the foreground color. *FBKeyboard()* terminates when the user presses either the new-line or return key (control-J or control-M).

To display the contents of a file in a viewport of the graphics display, KIC invokes the *FBMore()* routine. The *Textfile* argument is the file descriptor for the text file that is to be displayed in the viewport specified by the *Left*, *Bottom* and *Right*, *Top* display coordinates. *FBMore()* will display characters from the file until it has filled the space of the viewport, and then will prompt the KIC user in the same viewport before continuing; the procedure is similar to the UNIX *more(1)* command. As for the the *FBKeyboard()* routine, special hacking of control characters should be performed because a frame buffer with a serial, full duplex, RS-232 interface could enter an unknown state when it receives information that is similar to graphics command code. When the end of the file detected, the *FBMore()* routine will erase the viewport to the background color, color index equal to zero, and return. The file referenced by the *Textfile* argument is not closed before termination.

*FBPolygonClip()* will clip a polygon to the window defined by the *window* argument. The vertices of the polygon to be clipped are defined in the integer array referenced by the *xy* pointer; the number of vertices is specified by the integer referenced by the *ncoords* argument, and the first window coordinate is *(xy[0], xy[1])*, the second window coordinate is *(xy[2], xy[3])*, etc. It is assumed that the integer buffer referenced by the *xy* pointer is sufficiently large to contain the returned vertices of the clipped polygon. The number of vertices of the clipped polygon is returned in the integer referenced by the *ncoords* argument. See Section 4.1.3 that describes window-to-viewport clipping in KIC.

### 4.7. Geometry Display Routines

There are seven routines that are built on top of the frame buffer routines specifically for displaying objects in the layout viewports: *ShowBox()*, *ShowPolygon()*, *ShowWire()*, *ShowLabel()*, *ShowLine()*, *ShowManhattanLine()*, and *ShowPath()*. These routines perform the window-to-viewport transformations, window to viewport clipping, and then display the respective object in the appropriate viewport depending on the value of the *kpRedisplayControl* parameter. A synopsis of these display routines follows:

```
ShowBox(Layer, BB, Window)
int Layer;
struct ka BB, Window;

ShowLabel(Layer, Label, X, Y, Flag)
char *Label;
int Layer, X, Y, Flag;

ShowLine(Layer, X1, Y1, X2, Y2, Window)
int Layer, X1, Y1, X2, Y2;
struct ka Window;

ShowManhattanLine(Layer, X1, Y1, X2, Y2, Window)
int Layer, X1, Y1, X2, Y2;
struct ka Window;

ShowPath(Layer, Path, Window, Terminate)
struct p *Path;
struct ka Window;
int Layer, Terminate;

ShowPolygon(Layer, Path, Window)
struct p *Path;
struct ka Window;
int Layer;

ShowWire(Layer, Width, Path, Window)
struct p *Path;
struct ka Window;
int Layer, Width;
```

All coordinate values that are passed as arguments to these display routines must be world coordinates and not display coordinates. The *Layer* argument defines the color with which the object is to be displayed, and the *Window* argument defines the window to which the object should be clipped.

The display routines must also consider the fill pattern for the respective layer. Because there are frame buffers that do not provide the capability of stippled fill patterns, the KIC geometry display routines provide at least two modes of presentation for the objects on any layer. These two modes are filled and outlined, and the presentation mode for a given layer is determined by the logical value of the *klFilled* member of the KIC layer table for the respective layer. See Section 4.2.3 that describes the KIC layer table descriptor.

Except for a few peculiarities, these procedures should be easily understood by any programmer ho is familiar with the frame buffer routines and CD path descriptor. The remainder of section will discuss the peculiarities of the individual routines.

Because the CD database has no notion of text size, a *Window* argument is not passed to the *ShowLabel()* procedure. This display procedure will attempt to evaluate the size and position of the text label in display pixels from the *fFontWidth* and *fFontHeight* members of the frame buffer descriptor, and from this information it will determine the number of characters in the label that will fit into each of the layout viewports. Normally, the *ShowLabel()* routine will not attempt to display a label in a viewport if the number of display pixels to window lambda is less than two; the label will however always be displayed in all layout viewports if the *Flag* argument is set to a *True* logical value.

The *ShowManhattanLine()* routine is identical to the *ShowLine()* routine, except that it uses a more simple clipping algorithm. If this procedure is used to display a line that is neither vertical or horizontal with respect to the graphics display and is not contained entirely within the window defined by the *Window* argument, it will not be displayed accurately.

The *ShowPath()* routine will use the *FBLine()* procedure to display the path list referenced by the *Path* argument. If the *Terminate* argument is set, the path is terminated to produce the contour of a polygon.

### 4.7.1. Redisplay

Before an area of the layout viewports can be drawn, it must first be erased to the background color. The *ShowBox()* routine could be used to erase an area of the layout viewports by drawing a box on layer zero, but typically the *EraseBox()* procedure is used.

```
EraseBox(Area, Window)
struct ka Area, Window;
```

The *EraseBox()* procedure will erase the area identified by the *Area* argument in the window identified by the *Window* argument.

After an area of the layout viewports has been erased, the *Redisplay()* routine is invoked to display the layout information in the viewport.

```
Redisplay(SymbolDesc, AOI)
struct s *SymbolDesc;
struct ka AOI;
```

The *Redisplay()* procedure will display all layout information of the CD symbol referenced by the *SymbolDesc* symbol descriptor contained within the window defined by the *AOI* argument. The *kpRedisplayControl* member of the KIC parameters structure determines the viewports that will be affected by this routine. This routine is also controlled by the *ExpandInstances* and *kpExpandFineViewportOnly* members of the KIC parameters structure defined in Section 4.3.

Given the above geometry display routines, the *Redisplay()* procedure is a simple application of the CD database. The example C procedure that is provided in Section 2.6 as an example of traversing a symbol hierarchy is similar, but not identical, to the *Redisplay()* procedure.

Because of the display philosophy of KIC, in particular the fact that a red box displayed on top of a green box will hide the green box, this routine is inherently inefficient since it must traverse the entire symbol hierarchy once for each layer so that the layer interactions will be consistent (e.g., red objects are always displayed above green objects). There are at least two possible modifications that would increase the efficiency and speed of the *Redisplay()* procedure: the first is to modify the CD database to recognize the layers in the symbol's bin structure that are in use. This would eliminate the overhead of initializing CD to search an empty bin structure. The second modification would be to change the display philosophy of KIC and use the 'OR' ALU mode described in Section 4.6.2. In this case, each layer would correspond to one memory plane in the frame buffer, and a red object displayed on top of a green object would produce a, perhaps, brown object in the area of intersection. This approach was used in earlier versions of KIC but was rejected in favor of priority redisplay and color stipple patterns. The major problem with the plane-per-layer, color mixing solution is that it severely limits the number of uniquely colored layers that can be displayed simultaneously on the graphics display because most frame buffers have fewer than 10 memory planes, and most IC designers claim to be more comfortable with the notion of layer depth than with puzzling layer interactions.

### 4.8. Geometry Input Routines

There are seven routines in KIC that allow the user to create objects in the CD database. A synopsis of these procedures follows:

```
Arcs(LookedAhead)
int *LookedAhead;

Boxes(LookedAhead)
int *LookedAhead;

Doughnut(LookedAhead)
int *LookedAhead;

Flash(LookedAhead)
int *LookedAhead;

Label(LookedAhead)
int *LookedAhead;

Polygons(LookedAhead)
int *LookedAhead;

Wires(LookedAhead)
int *LookedAhead;
```

From the argument list of each routine, one could correctly conclude that these routines are similar. This section will describe the requirements and resulting similarities of these input procedures.

Each of the above procedures for creating geometries is invoked after the KIC user selects a corresponding command menu item. Each routine will terminate when the user either selects a new menu command or after an error condition is detected, such as if the CD database is unable to create an object because of a failure of the *malloc()* routine. While in one of the geometry creation routines, if the user selects another menu command, the integer that is referenced by the *LookedAhead* argument is set to the value of *True*. Another way of interpreting the use of the *LookedAhead* argument is that it is used by a child procedure to notify the parent procedure that the KIC user has performed an action that the child procedure is unable to handle. See Section 4.4 that describes the KIC command menus.

Because these procedures interface to the KIC user and thereby directly affect how efficiently he can use KIC, the procedures must be user-friendly. There is a set of requirements for any such routine. Firstly, geometry creation must be a mode of operation, rather than a single command; this allows the KIC user to create several objects without having to specify that he is doing so for each object. Secondly, the KIC user must be able to change the current layer while KIC is in the geometry creation mode of operation. The user must also be capable of 'undoing' his last action.

A pseudo C procedure is provided below as an example of a basic KIC user interface routine. All the above geometry creation routines are similar to the following model:

```
MakeObject( LookedAhead )
    int *LookedAhead;
    {
    initialize;
    use ShowPrompt to tell the user what he's doing;
    while( True ){
        Point();
        /* Has the user selected a menu command? */
        if ( Parameters.kpCommand[0] != EOS ){
            /* Did the user point to UNDO? */
            if ( strcmp(Parameters.kpCommand,MenuUNDO) == 0 ){
                if ( nothing to undo ){
                    *LookedAhead = True;
                    return;
                    }
                else if ( current object has been terminated ){
                    delete the entire object;
                    set flag to expect first point of new object;
                    }
                else{
                    undo the last point;
                    }
                }
            /* The user selected another menu command. */
            else{
                terminate the current object;
                *LookedAhead = True;
                return;
                }
            }
        /* Did the user not point in the layout viewport? */
        else if ( !Parameters.kpPointCoarseWindow )
            continue;

        /* Do we expect the first point of a new object? */
        else if ( expect first point ){
            begin new object;
            }

        /* Is this the same point as before? */
        else if ( Cursor.kcX == X And Cursor.kcY == Y ){
            terminate current object to begin another;
            set flag to expect first point of new object;
            }

        /* If none of the above, add point to object description. */
        else{
            add point to current object;
            }

        }
    }
```

The above *MakeObject()* procedure allows the user to describe an object by interactions with the graphical pointing device, and thereby enter the object into the CD database. The user also has the ability to 'undo' either the previously entered point or the previously created object. The procedure also recognizes when the user has not pointed in the layout viewports which permits the user to easily change the current layer by pointing in the layer menu viewport.

### 4.9. Geometry Modification Procedures

The KIC user must have the capability of easily editing or modifying geometric objects. There is an entire KIC command menu, the selection menu, dedicated to this user need. This chapter will describe KIC's philosophy of object modification and the user interface routines.

The first assumption of object editing in KIC is that the user will desire to modify several objects simultaneously rather than one object at a time. For example, he may wish to stretch a signal bus containing several wires, or he may want to copy a bipolar transistor and add a 90 degree rotation without destroying the inter-spacing of geometry.

The question then arises of how to remember and represent the several objects that the user will want to modify in a way that the user can easily control and in a way that will not be cumbersome for any editing operation. In other words, how would the KIC user define a working set of objects? One solution might be to require the user to define a rectangular area that contains every object to be modified. A problem with this simple solution, however, is that it does not allow the user to be as specific as he might wish to be; suppose, for example, that the user intends to move all objects in a rectangular area except one box.

The KIC solution to this problem is the selection queue that contains pointers to the user-specified objects in the symbol that is currently being edited. The selection queue is defined in the *select.h* file as follows:

```
struct ks {
      struct ks *ksSucc;
      struct o *ksPointer;
      };
struct ks *SelectQHead;
struct ka SelectQBB;
```

The following procedures are used by KIC to control the selection queue.

```
SelectQInit()

SelectQInsert(Pointer)
struct o *Pointer;

SelectQDelete(Pointer)
struct o *Pointer;

SelectQFirst(Pointer)
struct o **Pointer;

SelectQClear()

SelectQComputBB()

SelectQShow(AOI)
struct ka AOI;
```

The *SelectQInit()* procedure is invoked to initialiaze the selection mechanism. This procedure is invoked once by the *InitParameters()* procedure that assigns the default values to members of the KIC parameters structure.

*SelectQInsert()* is invoked to add an object that is referenced by *Pointer* to the selection queue. The object is always inserted at the top of the list. Any procedure that invokes *SelectQInsert()* is required by convention to set the information field of the particular object to the value of one via the *CDSetInfo()* database routine.

*SelectQDelete()* removes from the selection queue that object referenced by the *Pointer* argument and releases the memory that was used by the respective *ks* structure. No action occurs if the object descriptor is not contained in the selection queue. Any procedure that invokes *SelectQDelete()* is required by convention to set the information field of the removed object to the value of zero via the *CDSetInfo()* database routine.

The *SelectQFirst()* returns in the pointer referenced by *Pointer* the first object in the selection queue (the object that is at the head of the list). If the selection queue is empty, a null pointer is returned.

The *SelectQClear()* procedure is invoked to remove all objects from the selection queue and release all related memory. Before *SelectQClear()* terminates, the pointer to the top of the selection queue *SelectQHead* is assigned to the value of NULL.

When the *SelectQComputBB()* is invoked, the bounding box of all objects that are in the selection queue is evaluated and stored in the *SelectQBB* area descriptor. If the selection queue is empty, the resulting bounding box will have impossible values; the top of the box will be below the bottom, and the right side will be left of the left side.

The *SelectQShow()* routine is invoked to highlight in the layout viewports all objects contained in the selection queue that intersect the area defined by the *AOI* argument. The emphasis is provided by outlining the contour of all objects that qualify with the color that is identified by the *kpHighlightingPixel* member of the KIC parameters structure. This routine is always invoked by the *Redisplay()* procedure before it terminates.

The information field of an object is used by KIC to identify the status of the particular object. A table of the current uses of the object information field is shown below:

| | |
|---|---|
| Info = 0 | Object is unselected (default value). |
| *Info = 1 | Object is selected and in selection queue. |
| Info = 2 | Object is conditionally deleted and in selection queue. |
| Info = 3 | Object is conditionally copied and in selection queue. |
| Info = 4 | Object is conditionally selected and in selection queue. |
| Info = 5 | Object is polygon or wire being created. |
| Info = 10 | Object is box that could not be stretched (see modify.c) |
| *Info = 11-255 | Object has conditionally new layer and is in selection queue. The old layer number = Info - 10. |

* means that SelectQShow() will highlight these objects.

Given the above routines for managing the selection queue, an example is in order. The following procedure will examine the contents of the selection queue and delete all objects that are boxes:

```
#include "kic.h"
#include "select.h"

DeleteBoxes() {
     struct ks *SelectQDesc;
     char Type;

     SelectQDesc = SelectQHead;
     while(SelectQDesc != NULL){
          CDType(SelectQDesc->ksPointer,&Type);
          if(Type == CDBOX){
               CDDelete(Parameters.kpCellDesc,SelectQDesc->ksPointer);
               SelectQDelete(SelectQDesc->ksPointer);
               }
          SelectQDesc = SelectQDesc->ksSucc;
          }
     }
```

There are six routines for the user-editing of objects in KIC; each procedure uses the selection queue. A synopsis of each is provided below:

```
ChangeLayer(LookedAhead)
int *LookedAhead;

Copy(LookedAhead)
int *LookedAhead;

Del(LookedAhead)
int *LookedAhead;

Move(LookedAhead)
int *LookedAhead;

StretchBox(LookedAhead)
int *LookedAhead;

StretchPath(LookedAhead)
int *LookedAhead;
```

The requirements of these user-interface procedures for geometry modification are similar to those for geometry creation. The KIC user must be capable of 'undoing' any command, but he most also be capable of canceling the effect of an 'undo'. These procedures are also mode-oriented, thereby allowing the user to, for example, move objects in the layout viewports without being required to select the particular command before each action. Also, the user is capable of redefining the current transformation of objects at any time in the editing process; the current transformation is described in Section 4.3 covering the KIC parameters structure.

# Chapter 5

# System Dependencies

At present, KIC runs under Berkeley VM/UNIX, Masscomp Real-Time UNIX, and VAX/VMS; clearly the program is portable. The major problems that are involved with transporting KIC to another operating system are described in this chapter. It is, of course, impossible to predict every possible problem that may arise by attempting to move KIC to another system; only those difficulties that have been experienced in past are described. It is also assumed here that all C compilers are friendly and free of bugs and quirks!

The standard definition of the C programming language is [3]. It is assumed in KIC to be legal to pass entire structures in argument lists. All data structure member names in KIC are unique, and therefore no problem should ever arise from conflicting structure member names. Also, data unions are not used in KIC.

## 5.1. Terminal I/O Dependencies

Because the C programming language does not at present have a standard set of procedures for special I/O control, the first problem with transporting KIC to another system is usually the frame buffer interface. KIC must operate in a CBREAK mode; standard I/O, or *stdio*, is not sufficient. CBREAK mode implies that characters typed at the terminal are not buffered, but becomes available to the program when they are typed. Character echoing must also be disabled. This mode is easily obtained with the Berkeley UNIX *tty(4)* general tty interface; under VMS it is available from the *sys$qiow()* system service routine.

## 5.2. The Directory Search Path

To have the capability of easily using standard cell libraries, KIC must have a directory search path capability. When KIC requires that a file be opened, a list of directories will be searched for that file in the order that the directory names appear on the respective list. The first file found in the directory list is opened, and if the file name is not found, a new file is opened in the current directory. This directory search algorithm is provided in the *POpen()* routine.

The list of directories is defined by the *PSetPath()* routine that will invoke the *PConvertTilde()* procedure to perform special character conversion. Under Berkeley UNIX, the tilde (~) character is converted to the complete path name of the user's home directory; under VMS, the tilde character is converted to the user's login name.

To transport KIC to a system that does not have UNIX style directory path names or does not have the equivalent of the UNIX *getpwnam(3)* routine, it is necessary to modify the *PConvertTilde()* procedure such that it will perform tilde expansion correctly. And most importantly, the *POpen()* procedure must be modified such that it will be capable of correctly appending a file name to any directory name in the directory search list. As an example of this under the VMS file system, the file name *CELL.K* would have to be appended to the directory name *DMA0:[USER.JOE.LAYOUT]* to produce the character string *DMA0:[USER.JOE.LAYOUT]CELL.K* as the complete path name of the respective file.

### 5.3. Memory Management

On virtual memory systems, KIC and the CD database will allocate memory on demand. For speed considerations, a special memory management package called *nmalloc* has been developed. In general, the memory allocation procedure *malloc()*, which is provided in most C run-time libraries, attempts to be efficient and miserly with the existing free memory; consequently, it tends to be undesirably slow.

The *nmalloc()* routine maintains a separate free list for objects of a particular size. A free list in this case is a linked-list of free memory blocks; the first few bytes of the memory block are used as a pointer to the next free block of memory with the same size. Because large blocks of memory (greater than 100 bytes) are infrequently requested, free lists are maintained only for memory blocks smaller than 80 bytes. If a larger piece of memory is requested, the *nmalloc()* routine defaults to the usual *malloc()* memory allocation library routine.

All memory blocks are aligned by *nmalloc()* to the size of an integer, which is typically four bytes. Therefore, if *nmalloc()* is invoked to return a pointer to 10 bytes of free memory, the actual size of the allocated memory block will be 12 bytes. Furthermore, if *nmalloc()* maintains free lists for free memory blocks that are smaller than 80 bytes, only 20 free lists are required.

The free storage that is contained in a free list for a particular size of memory block is allocated by the *nmalloc()* routine only when a block of memory having that particular size is requested. When the free list must be constructed, or additional memory added to the list, *nmalloc()* will request a large block of memory from the system (typically 4096 bytes), and this memory block will then be divided to build the desired free list. Under Berkeley UNIX the *sbrk(2)* library routine is used to acquire this large block of memory from the system, and VMS uses the normal *malloc()* library routine.

To transport KIC to another system, the *nm_block_alloc()* procedure must be modified such that previously described free lists will be properly constructed. It is possible to compile KIC to use the usual *malloc()* library procedure, and not the *nmalloc* package, by setting the USE_OLD_MALLOC compiler flag in the *nmalloc.h* header file.

### 5.4. The System Interface

The *ShowProcess()* procedure is used by KIC to execute a system command or a child process. This routine must be capable of running a process, displaying any output in the area of the fine viewport, and detecting when the process has completed. Under Berkeley UNIX the *popen(3)* library routine is used, and under VMS the *LIB$SPAWN()* system service routine is used.

To transport KIC to another operating system, it is necessary to modify the *ShowProcess()* routine, and the argument lists must also be corrected. For example, to display a directory listing in the area of the fine viewport, the command string under UNIX is as follows:

```
ShowProcess( " ls -C " );
```

The corresponding command string under VMS is as follows:

```
ShowProcess( " DIRECTORY/COLUMN=3/OUTPUT=KIC " );
```

The *ShowPrompt()* procedure for VMS will recognize the "OUTPUT=KIC" string and replace the three characters "KIC" with the name of a temporary file. When the spawned process has completed, the contents of the temporary file are displayed in the area of the fine viewport.

# Appendix A

## A Catalog of All Routines and Macros

The following pages contain a catalog of all routines and macros used in KIC and the source files in which they are found. This list has two uses: it provides a index that will allow the programmer to quickly determine the source file that contains a particular routine, and it also allows the programmer to easily check the argument list of any function call.

| SYNOPSIS | SOURCE FILE |
|---|---|
| ABeginCall(SymbolNum)<br>int SymbolNum; | actions.c |
| ABeginSymbol(SymbolNum, A, B)<br>int SymbolNum;<br>int A, B; | actions.c |
| ABox(Length, Width, X, Y, XDirection, YDirection)<br>int Length, Width;<br>int X, Y;<br>int XDirection, YDirection; | actions.c |
| AComment(Text)<br>char *Text; | actions.c |
| ADeleteSymbol(SymbolNum)<br>int SymbolNum; | actions.c |
| AEnd() | actions.c |
| AEndCall() | actions.c |
| AEndSymbol() | actions.c |
| ALayer(Technology, Mask)<br>char Technology;<br>char *Mask; | actions.c |
| AMallocFailed() | actions.c |
| APolygon(Path)<br>struct p *Path; | actions.c |
| ARoundFlash(Width, X, Y)<br>int Width;<br>int X, Y; | actions.c |
| AT(Type, X, Y)<br>char Type;<br>int X, Y; | actions.c |
| AUserExtension(Digit, Text)<br>char Digit;<br>char *Text; | actions.c |
| AWire(Width, Path)<br>struct p *Path;<br>int Width; | actions.c |
| AddLayer() | attri.c |
| AddProperty() | prpty.c |
| AddResultingTransform(Pointer, TF)<br>struct o *Pointer;<br>int *TF; | move.c |
| AppendPointToPath(X, Y, Path)<br>struct p **Path;<br>int X, Y; | wires.c |

| SYNOPSIS | SOURCE FILE |
|---|---|
| Arcs(LookedAhead)<br>int *LookedAhead; | polygns.c |
| Area(LookedAhead)<br>int *LookedAhead; | select.c |
| Attri() | attri.c |
| BBLabel(Label, X, Y, BBCoarse, BBFine)<br>struct ka *BBCoarse;<br>struct ka *BBFine;<br>char *Label;<br>int X, Y; | labels.c |
| Basic() | basic.c |
| Blink(LookedAhead)<br>int *LookedAhead; | attri.c |
| Box(DisplayOrErase, Layer, L, B, R, T)<br>char DisplayOrErase;<br>int Layer, L, B, R, T; | boxes.c |
| Boxes(LookedAhead)<br>int *LookedAhead; | boxes.c |
| CDBB(SymbolDesc, Pointer, Left, Bottom, Right, Top)<br>struct s *SymbolDesc;<br>struct o *Pointer;<br>int *Left, *Bottom;<br>int *Right, *Top; | cd.c |
| CDBeginMakeCall(SymbolDesc, Name, NumX, DX, NumY, DY, Pointer)<br>struct s *SymbolDesc;<br>struct o *Pointer;<br>char *Name;<br>int NumX, DX, NumY, DY; | cd.c |
| CDAddProperty(SymbolDesc, Pointer, Value, String)<br>struct s *SymbolDesc;<br>struct o *Pointer;<br>char *String;<br>int Value; | cd.c |
| CDBox(Pointer, Layer, Length, Width, X, Y)<br>struct o *Pointer;<br>int *Length, *Width;<br>int *Layer;<br>int *X, *Y; | cd.c |
| CDCall(Pointer, SymbolName, NumX, DX, NumY, DY)<br>struct o *Pointer;<br>char *SymbolName;<br>int NumX, DX;<br>int NumY, DY; | cd.c |

| SYNOPSIS | SOURCE FILE |
|---|---|
| CDCheckPath(Path)<br>struct p *Path; | cd.c |
| CDClose(SymbolDesc)<br>struct s *SymbolDesc; | cd.c |
| CDDebug(Debug)<br>int Debug; | cd.c |
| CDDelete(SymbolDesc, ObjectDesc)<br>struct s *SymbolDesc;<br>struct o *ObjectDesc; | cd.h |
| CDDeleteObjectDesc(SymbolDesc, ObjectDesc)<br>struct s *SymbolDesc;<br>struct o *ObjectDesc; | cd.c |
| CDEndMakeCall(SymbolDesc, Pointer)<br>struct s *SymbolDesc;<br>struct o *Pointer; | cd.c |
| CDError(ID)<br>int ID; | cd.c |
| CDFrom(Root, CIFFile, A, B, Layers, NumLayers, Program)<br>char *Root, *CIFFile, Program;<br>int *Layers, NumLayers;<br>int A, B; | cd.c |
| CDGen(SymbolDesc, GenDesc, Pointer)<br>struct s *SymbolDesc;<br>struct g *GenDesc;<br>struct o **Pointer; | cd.c |
| CDGenCIF(FileDesc, SymbolDesc, SymbolNum, A, B, Program)<br>struct s *SymbolDesc;<br>FILE *FileDesc;<br>int *SymbolNum;<br>char Program;<br>int A, B; | cd.c |
| CDInfo(SymbolDesc, Pointer, Info)<br>struct s *SymbolDesc;<br>struct o *Pointer;<br>int *Info; | cd.c |
| CDInit() | cd.c |
| CDInitGen(SymbolDesc, Layer, Left, Bottom, Right, Top, GenDesc)<br>struct s *SymbolDesc;<br>struct g **GenDesc;<br>int Layer, Left, Bottom, Right, Top; | cd.c |
| CDInitTGen(Pointer, TGen)<br>struct o *Pointer;<br>struct t **TGen; | cd.c |

| SYNOPSIS | SOURCE FILE |
|---|---|
| CDInsertObjectDesc(SymbolDesc, ObjectDesc)<br>struct s *SymbolDesc;<br>struct o *ObjectDesc; | cd.c |
| CDIntersect(Left, Bottom, Right, Top, BeginX, EndX, BeginY, EndY)<br>int Left, Bottom, Right, Top;<br>int *BeginX, *EndX, *BeginY, *EndY; | cd.c |
| CDLabel(Pointer, Layer, Label, X, Y)<br>struct o *Pointer;<br>char *Label;<br>int *Layer;<br>int *X, *Y; | cd.c |
| CDMakeBox(SymbolDesc, Layer, Length, Width, X, Y, Pointer)<br>struct s *SymbolDesc;<br>struct o **Pointer;<br>int Layer, Length, Width, X, Y; | cd.c |
| CDMakeLabel(SymbolDesc, Layer, Label, X, Y, Pointer)<br>struct s *SymbolDesc;<br>struct o **Pointer;<br>char *Label;<br>int Layer;<br>int X, Y; | cd.c |
| CDMakePolygon(SymbolDesc, Layer, Path, Pointer)<br>struct s *SymbolDesc;<br>struct p *Path;<br>struct o **Pointer;<br>int Layer; | cd.c |
| CDMakeRoundFlash(SymbolDesc, Layer, Width, X, Y, Pointer)<br>struct s *SymbolDesc;<br>struct o **Pointer;<br>int Layer, Width, X, Y; | cd.c |
| CDMakeWire(SymbolDesc, Layer, Width, Path, Pointer)<br>struct s *SymbolDesc;<br>struct p *Path;<br>struct o **Pointer;<br>int Layer, Width; | cd.c |
| CDOpen(SymbolName, SymbolDesc, Access)<br>struct s **SymbolDesc;<br>char *SymbolName;<br>char Access; | cd.c |
| CDParseCIF(Root, CIFFile, Program)<br>char *Root, *CIFFile;<br>char Program; | cd.c |
| CDPatchInstances(SymbolDesc, MasterName)<br>struct s *SymbolDesc;<br>char *MasterName; | cd.c |

| SYNOPSIS | SOURCE FILE |
|---|---|
| CDPath(Path)<br>char *Path; | cd.c |
| CDPolygon(Pointer, Layer, Path)<br>struct o *Pointer;<br>struct p *Path;<br>int *Layer; | cd.c |
| CDProperty(SymbolDesc, Pointer, Property)<br>struct s *SymbolDesc;<br>struct o *Pointer;<br>struct prpty **Property; | cd.c |
| CDReflect(SymbolDesc)<br>struct s *SymbolDesc; | cd.c |
| CDRemoveProperty(SymbolDesc, Pointer, Value)<br>struct s *SymbolDesc;<br>struct o *Pointer;<br>int Value; | cd.c |
| CDRoundFlash(Pointer, Layer, Width, X, Y)<br>struct o *Pointer;<br>int *Layer, *Width, *X, *Y; | cd.c |
| CDSetInfo(SymbolDesc, Pointer, Info)<br>struct s *SymbolDesc;<br>struct o *Pointer;<br>int Info; | cd.c |
| CDSetLayer(Layer, Technology, Mask)<br>int Layer;<br>char Technology, *Mask; | cd.c |
| CDSymbol(SymbolName, SymbolDesc)<br>struct s *SymbolDesc;<br>char *SymbolName; | cd.c |
| CDT(Pointer, Type, X, Y)<br>struct o *Pointer;<br>char Type;<br>int X, Y; | cd.c |
| CDTGen(TGen, Type, X, Y)<br>struct t **TGen;<br>char *Type;<br>int *X, *Y; | cd.c |
| CDTo(CIFFile, Root, A, B, Program)<br>char *CIFFile, *Root;<br>char Program;<br>int A, B; | cd.c |
| CDType(Pointer, Type)<br>struct o *Pointer;<br>char *Type; | cd.c |

| SYNOPSIS | SOURCE FILE |
|---|---|
| CDUnmark(SymbolDesc)<br>struct s *SymbolDesc; | cd.c |
| CDUpdate(SymbolDesc, SymbolFile)<br>struct s *SymbolDesc;<br>char *SymbolFile; | cd.c |
| CDWire(Pointer, Layer, Width, Path)<br>struct o *Pointer;<br>struct p *Path;<br>int *Layer, *Width; | cd.c |
| Catch(sig)<br>int sig; | kic.c |
| CatchLyra(sig)<br>int sig; | lyra.c |
| CatchSIGINT(sig)<br>int sig; | kic.c |
| CenterFullView() | basic.c |
| ChangeLayer(LookedAhead)<br>int *LookedAhead; | change.c |
| ClipToGridPoint(X, Y)<br>int *X, *Y; | coords.c |
| Copy(LookedAhead)<br>int *LookedAhead; | copy.c |
| CopyPathWithXForm(Path)<br>struct p **Path; | copy.c |
| CtrlAt(Menu, X, Y)<br>char **Menu;<br>int X, Y; | point.c |
| Debug() | debug.c |
| DefaultWindows() | init.c |
| Del(LookedAhead)<br>int *LookedAhead; | delete.c |
| Desel() | select.c |
| Dimen(LookedAhead)<br>int *LookedAhead; | attri.c |
| DotKIC() | dotkic.c |
| Doughnut(LookedAhead)<br>int *LookedAhead; | polygns.c |
| Edit(Ready, Center, Modified)<br>int Ready, Center, Modified; | basic.c |

| SYNOPSIS | SOURCE FILE |
|---|---|
| ElapsedRealTime() | measure.c |
| ElapsedSystemTime() | measure.c |
| ElapsedUserTime() | measure.c |
| EraseBox(BB, Window)<br>struct ka BB, Window; | boxes.c |
| EraseLabel(Layer, Label, X, Y)<br>char *Label;<br>int Layer, X, Y; | labels.c |
| EraseMagnifyingGlass() | viewports.c |
| ErasePrompt() | viewports.c |
| Expand() | basic.c |
| FBBegin(Display)<br>char *Display; | fb.c |
| FBBlink(ColorId,Red,Green,Blue,Flag)<br>int ColorId, Flag;<br>int Red,Green,Blue; | fb.c |
| FBBox(ColorId, DisplayOrErase, Type, StyleId, Left, Bottom, Right, Top)<br>char Type, DisplayOrErase;<br>int StyleId, ColorId, Left, Bottom, Right, Top; | fb.c |
| FBDefineFillPattern(StyleId,BitArray)<br>int StyleId;<br>int *BitArray; | fb.c |
| FBDrawLineTo(X2,Y2)<br>int X2,Y2; | fb.c |
| FBEnd() | fb.c |
| FBFlood() | fb.c |
| FBForeground(DisplayOrErase, ColorId)<br>char DisplayOrErase;<br>int ColorId; | fb.c |
| FBHalt() | fb.c |
| FBInitialize() | |
| FBKeyboard(TypeIn)<br>char **TypeIn; | fb.c |
| FBLine(X1,Y1,X2,Y2)<br>int X1,Y1,X2,Y2; | fb.c |
| FBMore(Left,Bottom,Right,Top,Textfile)<br>int Left,Bottom,Right,Top;<br>FILE *Textfile; | fb.c |
| FBMoveTo(X1,Y1)<br>int X1,Y1; | fb.c |

| SYNOPSIS | SOURCE FILE |
|---|---|
| FBPoint(X,Y,Key,Buttons)<br>int *X,*Y,*Buttons;<br>char *Key; | fb.c |
| FBPolygon(ColorId, Type, StyleId, xy, ncoords)<br>int *xy;<br>int ColorId, StyleId, ncoords;<br>char Type; | fb.c |
| FBPolygonClip(xy, ncoords, window)<br>int *xy;<br>int *ncoords;<br>struct ka window; | fb.c |
| FBSetCursorColor(colorId)<br>int colorId; | fb.c |
| FBSetFillPattern(StyleId)<br>int StyleId; | fb.c |
| FBText(Mode, RowOrX, ColumnOrY, Text)<br>char *Text;<br>char Mode;<br>int RowOrX, ColumnOrY; | fb.c |
| FBTransfer() | fb.c |
| FBVLT(ColorId, R, G, B)<br>int ColorId, R, G, B; | fb.c |
| Fille(LookedAhead)<br>int *LookedAhead; | attri.c |
| FinePosition(X, Y, Key)<br>int X, Y;<br>char Key; | point.c |
| Flash(LookedAhead)<br>int *LookedAhead; | polygns.c |
| Flatten() | flatten.c |
| FlattenCell(CellDesc)<br>struct s *CellDesc; | flatten.c |
| FullRedisplay() | point.c |
| GenBeginCall(FileDesc, Number)<br>FILE *FileDesc;<br>int Number; | gencif.c |
| GenBeginSymbol(FileDesc, SymbolNum, A, B)<br>FILE *FileDesc;<br>int SymbolNum, A, B; | gencif.c |
| GenBox(FileDesc, Length, Width, X, Y, XDir, YDir)<br>FILE *FileDesc;<br>int Length, Width;<br>int X, Y, XDir, YDir; | gencif.c |

| SYNOPSIS | SOURCE FILE |
|---|---|
| GenComment(FileDesc, Text)<br>FILE *FileDesc;<br>char *Text; | gencif.c |
| GenEnd(FileDesc)<br>FILE *FileDesc; | gencif.c |
| GenEndCall(FileDesc)<br>FILE *FileDesc; | gencif.c |
| GenEndSymbol(FileDesc)<br>FILE *FileDesc; | gencif.c |
| GenLayer(FileDesc, Technology, Mask)<br>FILE *FileDesc;<br>char *Mask;<br>char Technology; | gencif.c |
| GenMirrorX(FileDesc)<br>FILE *FileDesc; | gencif.c |
| GenMirrorY(FileDesc)<br>FILE *FileDesc; | gencif.c |
| GenPolygon(FileDesc, Path)<br>FILE *FileDesc;<br>struct p *Path; | gencif.c |
| GenRotation(FileDesc, X, Y)<br>FILE *FileDesc;<br>int X, Y; | gencif.c |
| GenTranslation(FileDesc, X, Y)<br>FILE *FileDesc;<br>int X, Y; | gencif.c |
| GenUserExtension(FileDesc, Digit, Text)<br>FILE *FileDesc;<br>char *Text;<br>char Digit; | gencif.c |
| GenWire(FileDesc, Width, Path)<br>FILE *FileDesc;<br>struct p *Path;<br>int Width; | gencif.c |
| GetCurrentTransform() | copy.c |
| GetKeyWord(file, inbuf)<br>FILE *file;<br>char *inbuf; | dotkic.c |
| GetMoveTransform() | move.c |
| InBox(X, Y, AOI)<br>struct ka AOI;<br>int X, Y; | boxes.c |

| SYNOPSIS | SOURCE FILE |
|---|---|
| InPath(Delta, Path, X, Y)<br>struct p *Path;<br>int X, Y, Delta; | polygns.c |
| Init() | init.c |
| InitCoarseWindow(X, Y, Width)<br>int X, Y, Width; | init.c |
| InitFineWindow(X, Y)<br>int X, Y; | init.c |
| InitParameters() | init.c |
| InitSignals() | kic.c |
| InitVLT() | init.c |
| InitViewport() | init.c |
| Instances(LookedAhead)<br>int *LookedAhead; | instance.c |
| IsManhattan(X1, Y1, X2, Y2)<br>int X1, Y1, X2, Y2; | 45s.c |
| KIC() | viewports.c |
| LRCLayer(CellDesc,AOI,Layer)<br>struct s *CellDesc;<br>struct ka AOI;<br>int Layer; | lyra.c |
| LToP(Viewport, Window, X, Y)<br>struct ka Viewport;<br>struct ka Window;<br>int *X, *Y; | coords.h |
| Label(LookedAhead)<br>int *LookedAhead; | labels.c |
| LastView() | zoom.c |
| Lyra(LookedAhead)<br>int *LookedAhead; | lyra.c |
| MakeLayerInvisible(Layer)<br>int Layer; | attri.c |
| MakeLayerVisible(Layer)<br>int Layer; | attri.c |
| MallocFailed() | kic.c |
| MenuDeselect(Selection)<br>char *Selection; | viewports.c |
| MenuSelect(Selection)<br>char *Selection; | viewports.c |

| SYNOPSIS | SOURCE FILE |
|---|---|
| Move(LookedAhead)<br>int *LookedAhead; | move.c |
| MovePath(TX, TY, Path)<br>struct p **Path;<br>int TX, TY; | move.c |
| NextCellName() | kic.c |
| NotPointingAtLayout() | point.c |
| OutlineText(Left, Bottom, Right, Top, Type, DisplayOrErase, ColorId)<br>int Left, Bottom, Right, Top, ColorId;<br>char Type, DisplayOrErase; | viewports.c |
| OversizeBox(BB, Delta)<br>struct ka *BB;<br>int Delta; | boxes.c |
| PBox() | parser.c |
| PCIF(CIFFileName, StatusString, StatusInt)<br>char *CIFFileName;<br>char **StatusString;<br>int *StatusInt; | parser.c |
| PCall() | parser.c |
| PCharacter(Returned, WhiteSpaceControl, EOFControl)<br>int Returned, WhiteSpaceControl, EOFControl; | parser.h |
| PComment() | parser.c |
| PConvertTilde(psource, pdest, size)<br>char **psource;<br>char **pdest;<br>int *size; | paths.c |
| PDeleteSymbol() | parser.c |
| PEnd() | parser.c |
| PError(PErrorMessage)<br>char *PErrorMessage; | parser.c |
| PErrorCD() | parser.c |
| PErrorEOF() | parser.c |
| PErrorNoSemicolon() | parser.c |
| PErrorUndefinedLayer(Tech, Mask)<br>char *Mask;<br>char Tech; | parser.c |
| PGetPath() | paths.c |
| PInteger(Returned, EOFControl)<br>int Returned, EOFControl; | parser.h |

| SYNOPSIS | SOURCE FILE |
|---|---|
| PLayer() | parser.c |
| PLookAhead(Returned, WhiteSpaceControl, For)<br>int Returned, WhiteSpaceControl, For; | parser.h |
| PLookForSemi(Returned)<br>int Returned; | parser.h |
| POpen(file, mode, ext, prealname)<br>char *file;<br>char *mode;<br>char *ext;<br>char **prealname; | paths.c |
| PPath(Path)<br>struct p *Path; | parser.c |
| PPoint(X, Y)<br>int X, Y; | parser.c |
| PPolygon() | parser.c |
| PPrimitiveCommand() | parser.c |
| PRoundFlash() | parser.c |
| PSetPath(string)<br>char *string; | paths.c |
| PSymbol() | parser.c |
| PToL(Viewport, Window, X, Y)<br>struct ka Viewport;<br>struct ka Window;<br>int *X, *Y; | coords.c |
| PUserExtension() | parser.c |
| PWhiteSpace(Returned, WhiteSpaceControl, EOFControl)<br>int Returned, WhiteSpaceControl, EOFControl; | parser.h |
| PWhiteSpace1(Returned, EOFControl)<br>int Returned, EOFControl; | parser.h |
| PWhiteSpace2(Returned, EOFControl)<br>int Returned, EOFControl; | parser.h |
| PWhiteSpace3(Returned, EOFControl)<br>int Returned, EOFControl; | parser.h |
| PWire() | parser.c |
| Pan(LookedAhead)<br>int *LookedAhead; | zoom.c |
| Peek() | basic.c |
| PeekLayer(CellDesc, AOI, Layer, BottomVisibleLayer)<br>struct s *CellDesc;<br>struct ka AOI;<br>int Layer; | redisplay.c |

| SYNOPSIS | SOURCE FILE |
|---|---|
| int BottomVisibleLayer; | |
| Point() | point.c |
| PointLayerTable() | viewports.c |
| Point_At_LAYER() | select.c |
| Polygons(LookedAhead)<br>int *LookedAhead; | polygns.c |
| Pop() | contexts.c |
| PrintLRC(file)<br>FILE *file; | lyra.c |
| Properties(LookedAhead)<br>int *LookedAhead; | prpty.c |
| Push() | contexts.c |
| Redisplay(CellDesc, AOI)<br>struct s *CellDesc;<br>struct ka AOI; | redisplay.c |
| RedisplayAfterInterrupt() | redisplay.c |
| RedisplayLayer(CellDesc, AOI, Layer, BottomVisibleLayer)<br>struct s *CellDesc;<br>struct ka AOI;<br>int Layer;<br>int BottomVisibleLayer; | redisplay.c |
| RemoveLastPointInPath(Path)<br>struct p **Path; | wires.c |
| RemoveLayer(LookedAhead)<br>int *LookedAhead; | attri.c |
| RemoveProperty() | |
| RemovePropertyList(Pointer, PrptyDesc)<br>struct o *Pointer;<br>struct prpty **PrptyDesc; | prpty.c |
| RestorePropertyList(Pointer, PrptyDesc)<br>struct o *Pointer;<br>struct prpty *PrptyDesc; | prpty.c |
| Save() | basic.c |
| SaveDotKIC() | kic.c |
| SaveLastView() | zoom.c |
| SaveViewOnStack() | zoom.c |
| Sel(LookedAhead)<br>int *LookedAhead; | select.c |

| SYNOPSIS | SOURCE FILE |
|---|---|
| SelectQClear() | select.c |
| SelectQComputeBB() | select.c |
| SelectQDelete(Pointer)<br>struct o *Pointer; | select.c |
| SelectQFirst(Pointer)<br>struct o *Pointer; | select.c |
| SelectQInit() | select.c |
| SelectQInsert(Pointer)<br>struct o *Pointer; | select.c |
| SelectQShow(AOI)<br>struct ka AOI; | select.c |
| Selection(AOI)<br>struct ka AOI; | select.c |
| SelectionInstances(AOI)<br>struct ka AOI; | select.c |
| SelectionLayer(AOI, Layer)<br>struct ka AOI;<br>int Layer; | select.c |
| Selections(LookedAhead)<br>int *LookedAhead; | select.c |
| SetDebounceTime() | point.c |
| SetGridParameters(LookedAhead)<br>int *LookedAhead; | attri.c |
| SetMenuParameters(LookedAhead)<br>int *LookedAhead; | attri.c |
| ShowAttributeMenu() | attri.c |
| ShowAxes(Viewport, Window)<br>struct ka Viewport, Window; | grid.c |
| ShowBasicMenu() | basic.c |
| ShowBox(Layer, BB, Window)<br>int Layer;<br>struct ka BB, Window; | boxes.c |
| ShowCommandMenu() | viewports.c |
| ShowDebugMenu() | debug.c |
| ShowFineViewport() | viewports.c |
| ShowGrid(Viewport, Window, AOI)<br>struct ka Viewport, Window, AOI; | grid.c |
| ShowInstanceMenu() | instance.c |

| SYNOPSIS | SOURCE FILE |
|---|---|
| ShowLabel(Layer, Label, X, Y, Flag)<br>char *Label;<br>int Layer;<br>int X, Y;<br>int Flag; | labels.c |
| ShowLayerTable() | viewports.c |
| ShowLayout() | viewports.c |
| ShowLine(Layer, X1, Y1, X2, Y2, Window)<br>int Layer;<br>int X1, Y1, X2, Y2;<br>struct ka Window; | lines.c |
| ShowManhattanLine(Layer, X1, Y1, X2, Y2, Window)<br>int Layer;<br>int X1, Y1, X2, Y2;<br>struct ka Window; | lines.c |
| ShowMenu(Menu, NumMenu)<br>char **Menu;<br>int NumMenu; | viewports.c |
| ShowParameters() | viewports.c |
| ShowPath(Layer, Path, Window, Terminate)<br>struct p *Path;<br>struct ka Window;<br>int Layer;<br>int Terminate; | polygns.c |
| ShowPolygon(Layer, Path, Window)<br>struct p *Path;<br>struct ka Window;<br>int Layer; | polygns.c |
| ShowPrompt(Prompt)<br>char *Prompt; | viewports.c |
| ShowPromptAndWait(Prompt)<br>char *Prompt; | viewports.c |
| ShowProperties() | prpty.c |
| ShowPropertyMenu() | prpty.c |
| ShowRatio(Name, Value, PerUnitName, PerUnitValue)<br>char *Name;<br>char *PerUnitName;<br>int Value;<br>int PerUnitValue; | measure.c |
| ShowRGB() | attri.c |
| ShowSelectionMenu() | select.c |

| SYNOPSIS | SOURCE FILE |
|---|---|
| ShowWire(Layer, Width, Path, Window)<br>struct p *Path;<br>struct ka Window;<br>int Layer, Width; | wires.c |
| ShowXY() | viewports.c |
| StartTiming() | measure.c |
| StopTiming() | measure.c |
| StretchBox(LookedAhead)<br>int *LookedAhead; | modify.c |
| StretchPath(LookedAhead)<br>int *LookedAhead; | modify.c |
| SwapInts(Dragon, Eagle)<br>int Dragon, Eagle; | macros.h |
| SwitchToFinePositioning() | init.c |
| TCurrent(TFP)<br>int *TFP; | xforms.c |
| TEmpty() | xforms.c |
| TFull() | xforms.c |
| TIdentity() | xforms.c |
| TInit() | xforms.c |
| TInverse() | xforms.c |
| TInversePoint(X, Y)<br>int *X, *Y; | xforms.c |
| TMX() | xforms.c |
| TMY() | xforms.c |
| TPoint(X, Y)<br>int *X, *Y; | xforms.c |
| TPop() | xforms.c |
| TPremultiply() | xforms.c |
| TPush() | xforms.c |
| TRotate(XDirection, YDirection)<br>int XDirection, YDirection; | xforms.c |
| TTranslate(X, Y)<br>int X, Y; | xforms.c |
| To45(x1, y1, x2, y2)<br>int x1, y1, *x2, *y2; | 45s.c |
| Trap(n)<br>int n; | kic.c |

| SYNOPSIS | SOURCE FILE |
|---|---|
| TypeCoordinate() | point.c |
| UseLRC(cp)<br>char *cp; | lyra.c |
| UseRules(rules)<br>char *rules; | lyra.c |
| Visib(LookedAhead)<br>int *LookedAhead; | attri.c |
| WhereAmI(X, Y, Key)<br>int X, Y;<br>char Key; | point.c |
| Width(LookedAhead)<br>int *LookedAhead; | wires.c |
| Windo(LookedAhead)<br>int *LookedAhead; | zoom.c |
| Wires(LookedAhead)<br>int *LookedAhead; | wires.c |
| WriteCell() | contexts.c |
| Zoom(LookedAhead)<br>int *LookedAhead; | zoom.c |
| abs(Dragon)<br>int Dragon; | macros.h |
| free(ptr)<br>char *ptr; | nmalloc.h |
| index(s, c)<br>char *s, c; | paths.c |
| main(argc, argv)<br>int argc;<br>char *argv[]; | kic.c |
| max(Dragon, Eagle)<br>int Dragon, Eagle; | macros.h |
| min(Dragon, Eagle)<br>int Dragon, Eagle; | macros.h |

# Appendix B

# The CD Programmers's Manual

The Section 3 UNIX manual pages for the CD database package are contained in this appendix.

## NAME

CD - A Package of C Procedures for Managing CIF Databases

## SYNOPSIS

#include "cd.h"

## DESCRIPTION

*CD* is a package of C procedures for managing CIF (Caltech Intermediate Form) databases at an object level instead of the standard file level. For a description of the CIF language, see chapter four of the book by Mead and Conway entitled **Introduction to VLSI Systems** and also the Xerox PARC technical report by Hon and Sequin entitled **A Guide to LSI Implementation.** The reader should also be aware of the following terminology: *Master* refers to the definition of a symbol. *Instance* refers to a call of a symbol.

When a symbol is opened via *CDOpen*, it is mapped into main memory from files storing one symbol definition. Also, all referenced master symbols are read into main memory. A symbol that has been opened is referenced by a *symbol descriptor* defined below. To reflect at its secondary storage site the changes to a symbol that has been opened, invoke *CDUpdate*. To make an open symbol unknown to *CD*, invoke *CDClose*.

The types of valid objects within a symbol are CIF boxes, polygons, wires, roundflashes, and symbol calls or instance arrays. For each object, there is a procedure that creates the object in a particular symbol on a particular layer. See *CDMakePolygon*, *CDMakeWire*, *CDMakeBox*, and *CDMakeRoundFlash* below. The CIF *call* has been extended to handle instance arrays. To create an array, invoke *CDBeginMakeCall*, *CDT*, and *CDEndMakeCall*. All object creation procedures return a pointer to an *object descriptor* that can be used to reference the object later. *CDDelete* removes an object from a master cell.

*Bounding boxes* of every object can be accessed via *CDBB*. Along with each object is an integer *information field* that can be used for extending that object's description. See *CDSetInfo* and *CDInfo*. In addition to the integer information field, each object can have a linked list of property strings. See *CDAddProperty*, *CDRemoveProperty*, and *CDProperty*.

*CD* uses a *two dimensional, Manhattan transformation package* that the *CD* user may also invoke. The transformation package is discussed in detail below. With the transform package, you can define a current transformation, transform coordinates, and manage transformations with a transformation stack.

Traversing a symbol is done easily with a *generator loop*. To initialize a generator, invoke *CDInitGen* with an area of interest and layer of interest as parameters, and it will return a pointer to a *generator descriptor*. Every invocation of *CDGen* will then return a pointer to an object descriptor whose bounding box, transformed by the current transformation, intersects the area of interest and lies on the layer of interest. When *CDGen* has returned all qualifying objects, it returns a null pointer to the object descriptor.

*CDType* returns the type of an object descriptor, (e.g., box, polygon, etc.) and can be used to dispatch a type-specific procedure for manipulating the object.

To translate a CIF file into *CD* format, invoke *CDTo*. To translate a symbol into a CIF file, invoke *CDFrom*.

## ERROR HANDLING

*CD* has a simple mechanism for handling errors. In the file *cd.h*, there are two external error variables that are allocated by *CDInit*.

```
extern  int     CDStatusInt;
extern  char    *CDStatusString;
```

If a routine encounters any difficulty, it will place an identifying code in *CDStatusInt* and a pointer to diagnostic string in *CDStatusString*, and then return the with value of *False*. The possible fatal values for *CDStatusInt* are defined in the *cd.h* file as follows:

```
#define  CDPARSEFAILED       1    /* (FATAL) parse failed */
#define  CDNEWSYMBOL         3    /* symbol not in search path */
#define  CDMALLOCFAILED      11   /* (FATAL) out of memory */
#define  CDBADBOX            12   /* zero width or length box */
#define  CDXFORMSTACKFULL    13   /* transformation stack overflow */
#define  CDBADPATH           14   /* bad directory name search path */
```

Error handling in *CD* may be confusing at first because only those routines in which an error can occur will have a returned value. The routines in which no error is expected are assigned the type definition *void*.

## DESCRIPTOR TYPES

There are several descriptor types defined in the *cd.h* file. The four most frequently used are:

| descriptor type | structure name |
|---|---|
| symbol | s |
| object | o |
| generator | g |
| transform generator | t |

The definition of each descriptor is given below in the section concerning the *cd.h* file.

## INITIALIZATION

An application program must include the file *cd.h*.

**void CDInit()**

*CDInit* must be invoked before any other *CD* procedures. This routine will clear the layer table, set the directory search path to be the present working directory, initialize the transformation stack, and allocate storage for diagnostics.

**int CDPath(Path)**
**char *Path;**

*CDPath* sets the search rules for symbol-name resolution. *Path* is a pointer to a null terminated string containing a list of directory names to be searched separated by blanks. In the UNIX environment, *csh(1)* style names will be understood.

**void CDSetLayer(Layer,Technology,Mask)**
**int Layer;**
**char Technology,Mask[3];**

*CDSetLayer* tells *CD* that the layer *Layer* has the name *TechnologyMask*. The layer table is defined in the *cd.h* file as follows:

```
#define    CDNUMLAYERS        35

/* CD layer table */
struct l {
     char lTechnology;
     char lMask[3];
     /* True if CDFrom should output layer. */
     char lCDFrom;
     }
CDLayer[CDNUMLAYERS+1];
```

*CD* can be recompiled to provide a larger number of layers. Because layer numbers are stored in character fields, the absolute maximum number of layers is 255.

## SYMBOL MANAGEMENT

**int CDOpen(SymbolName,SymbolDesc,Access)**
**char *SymbolName,Access;**
**struct s **SymbolDesc;**

*CDOpen* opens a symbol named *SymbolName* and returns a pointer *SymbolDesc* to a symbol descriptor for it.

*Access* is a character that determines the result after the current search path has been examined for the existence of a symbol named *SymbolName*. If the character *Access* equals the character 'w', then *CDOpen* will create the cell in the database if it does not exist in the current search path. In other words, *CDOpen* will open the cell for writing. If *Access* equals the character 'r', then *CDOpen* will create the cell in the database if and only if it exists in the current search path. In other words, the symbol is only read into memory. If the cell does not exist in the current search path, no symbol is created in the database, and *SymbolDesc* is assigned the value of NULL. Finally, if *Access* equals the character 'n', the symbol is opened regardless of whether any symbol named *SymbolName* exists in the current search path. If such a file exists in the search path, it is not read into memory. In other words, CD creates a new symbol.

*CDOpen* will call a routine *PCIF* to read the symbol into the database. The user of *CD* must provide his own parser, and the parser is of course NOT required to understand only CIF; any language that is equivalent to CIF can be used. A synopsis of *PCIF* is as follows:

**PCIF(SymbolName, StatusString, StatusInt)**
**char *SymbolName;**
**char **StatusInt;**
**int *StatusInt;**

There are three requirements for *PCIF*. First, the parser must locate and read the symbol *SymbolName*, and insert the symbol definition into the *CD* database by using the object creation routines described below (e.g., *CDMakeBox*, *CDMakeWire*, etc.). Second, the parser must use a file called *parser.h* which contains the diagnostics described below. Finally, when the parse is completed, *PCIF* must return a pointer to a null terminated diagnostic string via *StatusString*, and *StatusInt* must be set to a value defined in the *parser.h* file as follows:

```
#define  PSUCCEEDED         1   /* successful return */
#define  PFAILED            2   /* parser failed */
#define  PNOTAPPLICABLE     3   /* parser failed due to syntax */
```

The diagnostic string *StatusString* may be NULL if and only if the parse succeeded.

*CDOpen* returns with the value *False* if the parse failed or if it was unable to allocate memory. When *CDOpen* returns, *CDStatusInt*, as defined above, is set to one of the values that are defined in the *cd.h* file as follows:

```
#define  CDPARSEFAILED      1   /* (FATAL) parse failed */
#define  CDOLDSYMBOL        2   /* symbol already in database */
#define  CDNEWSYMBOL        3   /* symbol not in search path */
#define  CDSUCCEEDED        4   /* new symbol(s) found in path */
```

If the return is fatal, *CDStatusString* contains a diagnostic message. Only CDPARSEFAILED is returned as a fatal error (i.e., *CDOpen* returns with the value *False );* this simplifies the diagnostic test. However, if the *Access* argument is set to 'r' and the symbol is not found in the search path, *CDOpen* returns with *CDStatusInt* set to CDNEWSYMBOL. The application programmer should be aware of this since it could be considered as or result in a fatal error.

**void CDSymbol(SymbolName,SymbolDesc)**
**char *SymbolName;**
**struct s **SymbolDesc;**

*CDSymbol* returns a pointer to symbol descriptor if the symbol *SymbolName* has been previously opened and exists in memory. If the symbol does not exist in memory, *SymbolDesc* is returned as a null pointer.

**int CDClose(SymbolDesc)**
**struct s *SymbolDesc;**

*CDClose* will remove the symbol identified by *SymbolDesc* from memory and any associated instances and geometries.

**int CDUpdate(SymbolDesc,SymbolName)**
**struct s *SymbolDesc;**
**char *SymbolName;**

*CDUpdate* will save any changes that have been made to the symbol referenced by *SymbolDesc*. The output will appear as CIF in a file identified by the null terminated string *SymbolName*. If *SymbolName* is a null pointer, the name of the CIF file will be identical to the name of the symbol being updated.

## CREATING GEOMETRIES

**int CDMakeBox(SymbolDesc,Layer,Length,Width,X,Y,Pointer)**
**struct s *SymbolDesc;**
**struct o **Pointer;**
**int Layer,Length,Width,X,Y;**

*CDMakeBox* will create a box of length *Length* in the x direction and width *Width* in the y direction, centered at *X,Y* on the layer *Layer* in the symbol identified by the descriptor *SymbolDesc*. *Pointer* contains a returned pointer to the object descriptor of the newly created box. *CDMakeBox* will return with the value *False* if it is unable to allocate storage. Otherwise, the value *True* is returned. Zero width or length boxes are not allowed.

**int CDMakeLabel(SymbolDesc,Layer,Label,X,Y,Pointer)**
**struct s *SymbolDesc;**
**struct o **Pointer;**
**int Layer;**
**char *Label;**
**int X,Y;**

*CDMakeLabel* will create a label on the layer *Layer* in the symbol identified by the descriptor *SymbolDesc*. The label will be referenced to the coordinate *X,Y*. *Label* is a pointer to a null terminated string containing the label. *Pointer* contains a returned pointer to the object descriptor of the newly created label. *CDMakeLabel* will return with the value *False* if it is unable to allocate storage. Otherwise, the value *True* is returned.

**int CDMakePolygon(SymbolDesc,Layer,Path,Pointer)**
**struct s *SymbolDesc;**
**struct o **Pointer;**
**struct p *Path;**
**int Layer;**

*CDMakePolygon* will create a polygon with a linked coordinate path *Path* on the layer *Layer* in the symbol identified by the descriptor *SymbolDesc*. *Path* is a pointer to a linked list of x,y coordinates that is defined in the *cd.h* file as follows:

```
/* Linked path structure */
struct p {
     int pX,pY;
     struct p *pSucc;
     };
```

*Pointer* contains a returned pointer to the object descriptor of the newly created polygon. *CDMakePolygon* will return with the value *False* if it is unable to allocate storage. Otherwise, the value *True* is returned.

**int CDMakeWire(SymbolDesc,Layer,Width,Path,Pointer)**
**struct s *SymbolDesc;**
**struct o **Pointer;**
**struct p *Path;**
**int Layer,Width;**

*CDMakeWire* will create a wire of width *Width* with a coordinate path *Path* on the layer *Layer* in the symbol referenced by the descriptor *SymbolDesc*. *Path* is a pointer to a linked list of x,y coordinates that is defined in the *cd.h* file as follows:

```
/* Linked path structure */
struct p {
     int pX,pY;
     struct p *pSucc;
     };
```

*Pointer* contains a returned pointer to the object descriptor of the newly created wire. *CDMakeWire* will return with the value *False* if it is unable to allocate storage. Otherwise, the value *True* is returned.

**int CDMakeRoundFlash(SymbolDesc,Layer,Width,X,Y,Pointer)**
**struct s *SymbolDesc;**
**struct o **Pointer;**
**int Layer,Width,X,Y;**

*CDMakeRoundFlash* will create a roundflash of diameter *Width* centered at *X,Y* on the layer *Layer* in the symbol identified by the descriptor *SymbolDesc*. *Pointer* contains a returned pointer to the object descriptor of the newly created roundflash. *CDMakeRoundFlash* will return with the value *False* if it is unable to allocate storage. Otherwise, the value *True* is returned. Zero diameter roundflashes are not allowed.

## CREATING INSTANCE ARRAYS

**int CDBeginMakeCall(SymbolDesc,SymbolName,NumX,DX,NumY,DY,Pointer)**
**struct s *SymbolDesc;**
**char *SymbolName;**
**struct o **Pointer;**
**int NumX,DX,NumY,DY;**

*CDBeginMakeCall* allocates memory and initializes the object descriptor that will represent the new symbol call to the symbol *SymbolName*. *NumX* is the number of instance in the untransformed x direction and *NumY* is the number of instances in the untransformed y direction. *DX* and *DY* are the distances between the left and right edges and the top and bottom edges respectively of two adjacent cells in the instance array. *Pointer* returns a pointer to the new object descriptor. If *SymbolName* is not in the current search path, or *CDBeginMakeCall* cannot allocate storage, *CDBeginMakeCall* returns with the value *False* and *CDStatusInt* will be set to one of the following values as defined in the *cd.h* file.

```
#define   CDPARSEFAILED       1     /* (FATAL) parse failed */
#define   CDNEWSYMBOL         3     /* symbol not in search path */
#define   CDMALLOCFAILED      11    /* (FATAL) out of memory */
```

**int CDT(Pointer,Type,X,Y)**
**struct o *Pointer;**
**char Type;**
**int X,Y;**

After invoking *BeginMakeCall*, *CDT* is invoked for each transformation in the call. *Pointer* is a pointer to the object descriptor that was returned by the call to *CDBeginMakeCall*. The character *Type* identifies the transformation to be added to the call. The valid arguments for *Type* are defined in the *cd.h* file as follows:

```
#define   CDMIRRORX        'x'    /* mirror in the direction of x */
#define   CDMIRRORY        'y'    /* mirror in the direction of y */
#define   CDROTATE         'r'    /* rotate by vector X,Y */
#define   CDTRANSLATE      't'    /* translate to X,Y */
```

The integers *X* and *Y* are used to qualify a rotation or translation. Only Manhattan rotations are valid. For a rotation of 90 degrees, *X* has the value of 0, and *Y* has the value of 1. For a rotation of 180 degrees, *X* has the value of -1, and *Y* has the value of 0. For a rotation of 270 degrees, *X* has the value of 0, and *Y* has the value of -1. For a translation, the integers *X* and *Y* are used to specify the x and y displacements, respectively.

Finally, *EndMakeCall* is invoked to insert the call into the master symbol identified by *SymbolDesc*. Remember that *transformation order is significant*.

**int CDEndMakeCall(SymbolDesc,Pointer)**
**struct s *SymbolDesc;**
**struct o *Pointer;**

*CDEndMakeCall* will insert the instance identified by the object descriptor that is pointed to by *Pointer* into the master symbol referenced by *SymbolDesc*. *Pointer* was returned by a previous call to *CDBeginMakeCall*. *CDEndMakeCall* will return with the value *False* if it is unable to allocate storage. Otherwise, the value *True* is returned.

## GENERATORS

**int CDInitGen(SymbolDesc,Layer,Left,Bottom,Right,Top,GenDesc)**
**struct s *SymbolDesc;**
**int Layer;**
**int Left,Bottom;**
**int Right,Top;**
**struct g **GenDesc;**

*CDInitGen* returns a pointer to a *generator storage descriptor* which is allocated automatically. Subsequent invocations of *CDGen* will return each geometry in the symbol identified by *SymbolDesc* on the layer *Layer* whose bounding box intersects the area of interest given by *Left*, *Bottom*, *Right*, and *Top*. If *Layer* equals zero, then the subsequent invocations of *CDGen* will return instances only (i.e., layer zero is the instance layer). *CDInitGen* will return with the value of *False* if it is unable to allocate the generator storage descriptor. Otherwise, the value of *True* is returned.

**void CDGen(SymbolDesc,GenDesc,Pointer)**
**struct s *SymbolDesc;**
**struct g *GenDesc;**
**struct o **Pointer;**

*CDGen* returns a pointer to an object descriptor which identifies an object within the area of interest as defined by the previous call to *CDInitGen* which returned the pointer to the descriptor *GenDesc*. If *CDGen* returns with *Pointer* set to NULL, then the last object has been returned and *GenDesc* storage has been freed.

**void CDType(Pointer,Type)**
**struct o *Pointer;**
**char *Type;**

*CDType* returns the type of an object pointed to by *Pointer*. This information routine is typically used in a generator loop to dispatch a type-specific procedure for accessing the object.

### ACCESSING GEOMETRIES

**void CDBox(Pointer,Layer,Length,Width,X,Y)**
**struct o *Pointer;**
**int *Layer,*Length,*Width,*X,*Y**

*CDBox* will return the length *Length* in the x direction and the width *Width* in the y direction of a box identified by the pointer *Pointer* to an object descriptor. The box is centered at the coordinate *X*, *Y* and on the layer *Layer*.

**void CDLabel(Pointer,Layer,Label,X,Y)**
**struct o *Pointer;**
**char **Label;**
**int *Layer;**
**int *X,*Y;**

*CDLabel* returns the pointer to a null terminated label *Label* that has lower, left justification to the coordinate *X*,*Y* and whose object descriptor is pointed to by *Pointer*. The label is on the layer *Layer*.

**void CDPolygon(Pointer,Layer,Path)**
**struct o *Pointer;**
**int *Layer;**
**struct p **Path;**

*CDPolygon* will return a pointer to the linked coordinate path *Path* of a polygon identified by the pointer *Pointer* to an object descriptor. The polygon is on the layer *Layer*. The linked coordinate path is defined above in the description of *CDMakePolygon*.

**void CDWire(Pointer,Layer,Width,Path)**
**struct o *Pointer;**
**struct p **Path;**
**int *Layer,*Width;**

> *CDWire* will return a pointer to the linked coordinate path *Path* of a wire with width *Width* that is identified by the pointer *Pointer* to an object descriptor. The wire is on the layer *Layer*.

**void CDRoundFlash(Pointer,Layer,Width,X,Y)**
**struct o *Pointer;**
**int *Layer,*Width,*X,*Y;**

> *CDRoundFlash* will return the diameter *Width* of a roundflash identified by the pointer *Pointer* to an object descriptor. The roundflash is centered at the coordinate *X, Y* and on the layer *Layer*.

### ACCESSING AN INSTANCE

**void CDCall(Pointer,SymbolName,NumX,DX,NumY,DY)**
**struct o *Pointer;**
**char **SymbolName;**
**int *NumX,*DX,*NumY,*DY;**

> *CDCall* returns the a character pointer to the name *SymbolName* of an instance referenced by the object descriptor pointed to by *Pointer*. Also returned is *NumX* which is the number of instance in the untransformed x direction and *NumY* which is the number of instances in the untransformed y direction. *DX* and *DY* are the distances between the left and right edges and the top and bottom edges respectively of two adjacent cells in the instance array.

**void CDInitTGen(Pointer,TGen)**
**struct o *Pointer;**
**struct t **TGen;**

> The pointer *Pointer* should point to an object descriptor of an instance. *CDInitTGen* initializes the transformation generator loop to access the transformations of an instance referenced by *Pointer*. *TGen* is a returned pointer to a *transform generator descriptor*. Subsequent invocations of *CDTGen* will return the instance transformations in order.

**void CDTGen(TGen,Type,X,Y)**
**struct t **TGen;**
**char *Type;**
**int *X,*Y;**

> *CDTGen* returns a pointer to a character which identifies a transformation of the instance for which *TGen* was returned by *CDInitTGen* as a pointer to the transform generator descriptor. The character pointer *Type* identifies a transformation that was added to the instance by a call to *CDT*. The order in which transformations are returned by *CDTgen* is identical to the order in which they were specified by calls to *CDT*. The possible returned values for *Type* are defined in the *cd.h* file as follows:

```
#define   CDMIRRORX      'x'   /* mirror in the direction of x */
#define   CDMIRRORY      'y'   /* mirror in the direction of y */
#define   CDROTATE       'r'   /* rotate by vector X,Y */
#define   CDTRANSLATE    't'   /* translate to X,Y */
```

The integer pointers *X,Y* are used to specify a rotation or translation. For a rotation of 90 degrees, *X* has the returned value of 0, and *Y* has the returned value of 1. For a rotation of 180 degrees, *X* has the returned value of -1, and *Y* has the returned value of 0. For a rotation of 270 degrees, *X* has the returned value of 0, and *Y* has the returned value of -1. For a translation, the returned values of *X* and *Y* specify the x and y displacements, respectively. If *CDTGen* returns with *TGen* set to NULL, then the last transformation has been returned and *TGen* storage has been freed.

**INFORMATION ROUTINES**

**int CDBB(SymbolDesc,Pointer,Left,Bottom,Right,Top)**
**struct s *SymbolDesc;**
**struct o *Pointer;**
**int *Left,*Bottom,*Right,*Top;**

*CDBB* returns the bounding box of an object pointed to by *Pointer* in the symbol identified by *SymbolDesc*. If *Pointer* is a null pointer, then *CDBB* returns the bounding box of the entire symbol.

*CDBB* may have to use temporary storage during the computation of a symbols bounding box. If it can not allocate the required memory, *CDBB* returns with the value of *False*. Otherwise, the value of *True* is returned.

**void CDInfo(SymbolDesc,Pointer,Info)**
**struct s *SymbolDesc;**
**struct o *Pointer;**
**int *Info;**

*CDInfo* returns the value *Info* of the info field of an object referenced by *Pointer*. If *Pointer* is a null pointer, then the value of the info field of the symbol referenced by *SymbolDesc* is returned.

**void CDSetInfo(SymbolDesc,Pointer,Info)**
**struct s *SymbolDesc;**
**struct o *Pointer;**
**int Info;**

*CDSetInfo* sets the info field the object pointed to by *Pointer*. If *Pointer* is a null pointer, then the info field of the symbol referenced by *SymbolDesc* is set.

```
void CDProperty(SymbolDesc,Pointer,Property)
struct s *SymbolDesc;
struct o *Pointer;
struct prpty **Property;
```

*CDProperty* returns the pointer *Property* which points to the linked list of properties associated with the object referenced by *Pointer*. If *Pointer* is a null pointer, then the property list of the symbol referenced by *SymbolDesc* is returned. The pointer *Property* is returned as a null pointer if there is no property list associated with the particular object.

The property list structure is defined in the *cd.h* file as follows:

```
/* Property List desc. */
struct prpty {
      int prpty_Value;
      char *prpty_String;
      struct prpty *prpty_Succ;
      };
```

A property consists of an identifying integer and a null terminated character string extension. The linked list of properties is terminated by a null *prpty_Succ* pointer. Properties are assigned to an object or symbol via the *CDAddProperty* routine.

```
int CDAddProperty(SymbolDesc,Pointer,Value,String)
struct s *SymbolDesc;
struct o *Pointer;
char *String;
int Value;
```

*CDAddProperty* inserts property information into the property list of the object pointed to by *Pointer*. If *Pointer* is a null pointer, then the property is added to the property list of the symbol referenced by *SymbolDesc*. The property information consists of an identifying integer *Value* and a null terminated character string extension that is pointed to by *String*. If *CDAddProperty* can not allocate memory, the value *False* is returned. Otherwise, the value of *True* is returned.

```
int CDRemoveProperty(SymbolDesc,Pointer,Value)
struct s *SymbolDesc;
struct o *Pointer;
int Value;
```

*CDRemoveProperty* deletes property information from the property list of the object pointed to by *Pointer*. If *Pointer* is a null pointer, then the property is removed from the property list of the symbol referenced by *SymbolDesc*. Every property with the value of *Value* is removed. If *CDRemoveProperty* has trouble allocating or releasing memory, the value *False* is returned. Otherwise, the value of *True* is returned.

### OBJECT DELETION

**void CDDelete(SymbolDesc,Pointer)**
**struct s *SymbolDesc;**
**struct o *Pointer;**

*CDDelete* will remove the object pointed to by *Pointer* from the symbol referenced by *SymbolDesc*.

### INTEGRITY

**int CDReflect(SymbolDesc)**
**struct s *SymbolDesc;**

*CDReflect* must be invoked at certain times by the *CD* application if the symbol that is referenced by *SymbolDesc* is modified. If the modification to the symbol has changed its bounding box, a call to *CDReflect* will update the bounding box information in every other symbol in the *CD* database that either directly or indirectly references it. The value of *False* is returned if *CDReflect* is unable to allocate new memory. Otherwise, the value of *True* is returned.

### TRANSLATING TO AND FROM CIF

**int CDTo(CIFFile,Root,A,B,Program)**
**char *CIFFile, *Root;**
**int A,B;**
**char Program;**

*CDTo* translates from a CIF file named *CIFFile* into symbol files, each having a file name identical to the symbol that it contains. CIF commands that are not between a **DS** and a matching **DF** are stored in file named *Root*. All objects are scaled by the ratio *A/B*.

*CDTo* will call a routine *PCIF* to read the input CIF file. The requirements of this parser are discussed above in the description of *CDOpen*.

Because each program embeds symbol names differently, the character *Program* will tell *CDTo* and the parser *PCIF* which program generated the CIF file. Before calling the parser, *CDTo* sets the *dProgram* character in the *CD* parameters structure to the character *Program*. By accessing this structure element, the parser determines the origin of the CIF. See the section on the *cd.h* file for a description of the *CD* parameter structure. The following arguments for *Program* are valid for the parser in *~cad/src /kic /parser.c :*

| **Program** | **CIF generator** |
|---|---|
| 'a' | Stanford CIF, |
| 'b' | NCA CIF, |
| 'h' | HP's IGS, |
| 'i' | Xerox's Icarus, |
| 'k' | Berkeley's KIC, |
| 'm' | Mextra-style CIF, |
| 's' | Xerox's Sif, |
| 'n' | none of the above. |

If *CDTo* encounters any difficulty in the CIF conversion, the value of *False* is retuned and *CDStatusInt* is set to value of *CDPARSEFAILED* and a diagnostic message is placed in *CDStatusString*.

**int CDFrom(Root,CIFFile,A,B,Layers,NumLayers,Program)**
**char *Root,*CIFFile,Program;**
**int *Layer,NumLayers;**
**int A,B;**

*CDFrom* translates a symbol hierarchy rooted with the symbol named *Root* into a CIF file named *CIFFile*. The style of CIF output is identified by the character *Program*. See the description of the *CDTo* procedure for a list of valid arguments for *Program*. All objects are scaled by the ratio *A/B*. All instances in the symbol hierarchy must exist in the current search path.

The *Layers* argument is a pointer to an array of *NumLayers* integers that are used to mask certain layers in CD layer table. If *Layers*[ **N** ] is zero, where **N** is a non-negative integer less than *NumLayers*, then any object that is on layer **N** will not appear in the CIF output.

If *CDFrom* encounters any difficulty in the conversion, the value of *False* is retuned and *CDStatusInt* is set to one of the following values defined in the *cd.h* file:

```
#define    CDPARSEFAILED      1     /* (FATAL) parse failed */
#define    CDNEWSYMBOL        3     /* symbol not in search path */
#define    CDMALLOCFAILED     11    /* (FATAL) out of memory */
```

If no difficulty is encountered, *CDFrom* returns with the value of *True*.

**int CDParseCIF(Root,CIFFile,Program)**
**char *Root,*CIFFile,Program;**

*CDParseCIF* will construct a *CD* database rooted at a symbol named *Root* from a CIF file *CIFFile* rather than a hierarchy of symbol files. The style of CIF input is identified by the character *Program*. See the description of the *CDTo* procedure for a list of valid arguments for *Program*.

When *CDParseCIF* encounters a reference to a layer that was not previously defined in the CD layer table by a call to *CDSetLayer*, the new layer is. added to the layer table. This differs from the *CDFrom* and *CDOpen* routines that will return CDPARSEFAILED whenever they encounter an undefined layer. A layer is considered undefined if the *lTechnology* field in the CD layer table is a blank character. See the above section that describes the initialization routines.

If *CDParseCIF* encounters any difficulty in the conversion, the value of *False* is retuned and *CDStatusInt* is set to one of the following values defined in the *cd.h* file:

```
#define    CDPARSEFAILED      1     /* (FATAL) parse failed */
#define    CDNEWSYMBOL        3     /* symbol not in search path */
#define    CDMALLOCFAILED     11    /* (FATAL) out of memory */
```

If no difficulty is encountered, *CDParseCIF* returns with the value of *True*.

## EXAMPLE

The following pseudo C routine is an example of traversing a symbol called *SymbolName*. Traversal in this case means to access the description of all objects that lie in an area of interest in a symbol. The computations performed with the accessed descriptions will depend of course on the application.

```
Traverse(SymbolName,Left,Bottom,Right,Top)
    char *SymbolName;
    int Left,Bottom,Right,Top;
    {
    struct s *SymbolDesc;
    struct o *Pointer;
    struct g *GenDesc;
    char *InstanceName;
    int NumX,NumY,DX,DY,X,Y,Layer;
    char Type;

    /* Open symbol named SymbolName (we assume here that it exists) */
    CDOpen(SymbolName,&SymbolDesc,'r');

    /*****************************************************************
    *
    * Traverse the instances first.
    * Layer # 0 is the instances layer, because it doesn't make
    * sense for an instance to be associated with a particular layer.
    *
    * First, initialize the generator loop so the generator will
    * return all instances whose bounding boxes intersect the
    * area of interest.
    *
    *****************************************************************/

    CDInitGen(SymbolDesc,0,Left,Bottom,Right,Top,&GenDesc);

    loop {
        /* Invoke CDGen to access a pointer to an instance array */
        CDGen(SymbolDesc,GenDesc,&Pointer);

        /* Have all instances have been traversed? */
        if(Pointer == NULL)
            break;

        /* Access the instance array */
        CDCall(Pointer,&InstanceName,&NumX,&DX,&NumY,&DY);
```

```
                /*******************************************************
                *
                * This instance array calls the symbol named InstanceName.
                * Recursively invoke Traverse to traverse InstanceName.
                *
                *******************************************************/
                Traverse(InstanceName,Left,Bottom,Right,Top);
                }

        /****************************************************************
        *
        * Now, traverse the geometries by layer.
        *
        ****************************************************************/
        for(Layer = 1; Layer <= CDNUMLAYERS; ++Layer) {

                /* Initialize the generator loop */
                CDInitGen(SymbolDesc,Layer,Left,Bottom,Right,Top,&GenDesc);

                loop {
                        /* Invoke CDGen to access pointer to an object */
                        CDGen(SymbolDesc,GenDesc,&Pointer);

                        /* Last geometry? */
                        if(Pointer == NULL)
                                break;

                        /* Access the type of the geometry as Type */
                        CDType(Pointer,&Type);

                        /* Dispatch according to Type to specific procedure */
                        if(Type == CDBox) {
                                /* Access box */
                                CDBox(Pointer,&Layer,...);
                                /* Compute with the box */
                                        .
                                        .
                                        .
                                }
```

```
            else if(Type == CDWire) {
                /* Access wire */
                CDWire(Pointer,&Layer,...);
                /* Compute with the wire */
                   .
                   .
                   .
                }
               .
               .
              etc.
               .
               .
            }
        }
    }
```

## STORAGE BINS

All objects contained within a symbol are stored in *bins*. A pointer to these storage bins is contained in the symbol descriptor, and each symbol has it's own storage bins. This section will present a brief description of the storage algorithm.

The world coordinate system in *CD* ranges from CDBINMINX to CDBINMAXX in the x-direction, and from CDBINMINY to CDBINMAXY where these values are defined in the *cd.h* file. The world coordinate system is covered by an array of CDNUMBINS by CDNUMBINS storage bins (the number if bins is always a perfect square). There is an additional storage bin which is called the ***residual bin***.

Each storage bin contains the starting pointer to a linked list of object descriptors, those objects which are contained in that particular bin. When an object is inserted into a bin as a result of a call to one of the object creation routines described above, it is inserted at the top of the linked list. The bin into which an object is inserted depends on the bounding box of that object. If the bounding box intersects an area covered by more than one bin, the object is inserted into the residual bin. Otherwise, the object is inserted into the bin that contains the bounding box of the object.

If *CD* is compiled with a large number of bins, it will be able to traverse a symbol hierarchy very rapidly. However, a large number of storage bins also means that a large amount of memory must be dedicated to the bin pointers. *CD* allocates the memory for storage bins on demand; this means that the memory requirements for the storage bins is also a function of the number of layers being used in the symbol.

The values of CDBINMAXX, CDBINMINX, CDBINMAXY, and CDBINMINY should be as small as possible while still reflecting a realistic working area for the *CD* application. This will insure that every storage bin will be used. *CD* will not fail if the real working area exceeds *CD's* range for the world coordinate system. Objects that exist outside the world coordinate system of *CD* will be inserted into the outer-most bins.

**PROCEDURES FOR TWO DIMENSIONAL TRANSFORMATIONS**

The following routines provide two dimensional transformations using integer arithmetic. The source to these routines can be found in *~cad/src/kic*. The package of routines includes such capabilities as translation, mirroring, rotation, a transformation stack, and inverse transformations. For a further discussion of these procedures, see Newman and Sproull's text **Principles of Interactive Computer Graphics** .

**INITIALIZATION OF THE TRANSFORM PACKAGE**

**void TInit()**

*TInit* initializes the transform package and returns with the current transform equal to the identity transform.

**THE CURRENT TRANSFORMATION**

**void TIdentity()**

*TIdentity* sets the current transform the identity matrix. The previous current transform is destroyed.

**void TTranslate(X,Y)**
**int X,Y;**

*TTranslate* postmultiplies the current transformation matrix by a transform that translates by a displacement of *X,Y* .

**void TMY()**

*TMY* postmultiplies the current transformation matrix by a transform that mirrors the y-coordinates (i.e., mirrors in the direction of the y axis).

**void TMX()**

*TMX* postmultiplies the current transformation matrix by a transform that mirrors the x-coordinates (i.e., mirrors in the direction of the x axis).

**void TRotate(XDirection,YDirection)**
**int XDirection,YDirection;**

*TRotate* postmultiplies the current transformation matrix by a transform that rotates counter-clockwise by an angle that is expressed as a CIF-style direction vector. Only 0, 90, 180, 270 degree rotations are allowed.

**void TPoint(X,Y)**
**int *X,*Y;**

*TPoint* transforms the point *X,Y* by multiplying it by the current transformation matrix.

**THE TRANSFORMATION STACK**

**int TEmpty()**

*TEmpty* returns the value of *True* if the transformation stack is empty. Otherwise, the value of *False* is returned.

**int TFull()**

*TFull* returns the value of *True* if the transformation stack is full. Otherwise, the value of *False* is returned.

**void TPush()**

*TPush* pushes the current transform onto the transformation stack. The value of the current transform is not changed. The transformation stack is not checked for an overflow condition.

**void TPop()**

*TPop* pops the current transform from the transformation stack. The value of the current transform becomes the transform that was most recently pushed onto the stack. The transformation stack is not checked for an underflow condition.

**void TCurrent(TF)**
**int *TF;**

*TCurrent* places the current transform matrix in a nine integer array that is passed from the calling program. The first row of the transformation matrix appears as the first three integers in the argument, the second row of the transformation matrix appears as the next three integers, etc. After several transformations have been defined by *TTranslate(), TRotate(), TMX(),* and *TMY(),* one can determine the minimum resultant or equivalent transformation through the examination of the elements of the current transformation matrix as described in the following table.

| TF[0] | TF[3] | TF[1] | TF[4] | Transformation |
|---|---|---|---|---|
| 1 | 0 | 0 | 1 | Translate only. |
| 0 | -1 | 1 | 0 | Rotate 90 deg., translate. |
| 0 | 1 | -1 | 0 | Rotate 180 deg., translate. |
| -1 | 0 | 0 | -1 | Rotate 270 deg., translate. |
| -1 | 0 | 0 | 1 | Mirror in X, translate. |
| 1 | 0 | 0 | -1 | Mirror in Y, translate. |
| 0 | -1 | -1 | 0 | Mirror in X, rotate 90 deg., translate. |
| 0 | 1 | 1 | 0 | Mirror in X, rotate 270 deg., translate. |

For all cases, the X,Y translation vector is given by TF[6], TF[7].

### THE INSTANCE TRANSFORMATION

**void TPremultiply()**

*TPremultiply* forms the instance transform by multiplying the current transform with the transform that was last pushed onto the transformation stack and placing the product in the current transformation matrix. Thus, the procedure for transforming the coordinates of an instance is demonstrated below:

```
/* push master cell transform onto stack */
TPush();

/* set current transform to identity */
TIdentity();

/* Invoke TMX, TTranslate, etc. to build instance transform */
TMX();
TTranslate(Dx,Dy);

/* Form the instance transform */
TPremultiply();

/* Invoke TPoint to transform instance points */
TPoint(&X,&Y);

/* return to master transform */
TPop();
```

### INVERSE OF CURRENT TRANSFORMATION

**void TInverse()**

*TInverse* computes the inverse transform of the current transform.

**void TInversePoint(X,Y)**
**int *X,*Y;**

*TInversePoint* transforms the point *X,Y* by multiplying it by the inverse transform matrix. A call to this routine must be preceded by a call to *TInverse*.

### ANOTHER EXAMPLE

To illustrate the use of the transform package, let us modify the routine *Traverse* shown earlier. The new routine is intended to display a symbol on a graphics terminal. When each object is accessed, it will be clipped to the area of interest and displayed.

```
main(){
          .
          .
          .
        /* initialize transformation stack */
        TInit();
          .
          .
          .
        /* display SymbolName in the area Left,Bottom - Right,Top */
        Display(SymbolName,Left,Bottom,Right,Top)
          .
          .
          .
        }

Display(SymbolName,Left,Bottom,Right,Top)
        char *SymbolName;
        int Left,Bottom,Right,Top;
        {
        struct s *SymbolDesc;
        struct g *GenDesc;
        struct o *Pointer;
        struct t *TGen;
        char *InstanceName;
        int NumX,NumY,DX,DY,X,Y,Layer;
        char Type;

        /* open symbol named SymbolName (here we assume it exists) */
        CDOpen(SymbolName,&SymbolDesc,'r');

        /* initialize generator to return instances in SymbolName */
        CDInitGen(SymbolDesc,0,Left,Bottom,Right,Top,&GenDesc);

        loop {
                /* Invoke CDGen to access a pointer to an instance array */
                CDGen(SymbolDesc,GenDesc,&Pointer);

                /* Have all instances been traversed? */
                if(Pointer == NULL)
                        break;

                /* push current transform onto stack */
                TPush();

                /* set current transform to identity */
                TIdentity();

                /* Access instance information */
                CDCall(Pointer,&InstanceName,&NumX,&DX,&NumY,&DY);
        /* Initialize generator to return transform of InstanceName */
        CDInitTGen(Pointer,&TGen);

        /* place instance transformation in current transformation */
        loop {
            CDTGen(&TGen, &Type, &X, &Y);
            if(TGen == NULL)
                break;
            else if(Type == CDTRANSLATE)
                TTranslate(X, Y);
            else if(Type == CDMIRRORX)
                TMX();
            else if(Type == CDMIRRORY)
                TMY();
            else if(Type == CDROTATE)
                TRotate(X, Y);
            }

        /* Combine transform of InstanceName with it's master */
        TPremultiply();

        /* recursively call display to traverse and display instance */
        Display(InstanceName,Left,Bottom,Right,Top);

        /* pop master transform from stack */
        TPop();
        }

    /* now traverse the geometries */
    for(Layer = 1; Layer <= CDNUMLAYERS; ++Layer) {

        /* Set the current color to be a color associated with Layer. */
        SetColor(Layer);

        /* initialize generator for layer Layer */
        CDInitGen(SymbolDesc,Layer,Left,Bottom,Right,Top,&GenDesc);

        loop {
            /* Invoke CDGen to access pointer to an object */
            CDGen(SymbolDesc,GenDesc,&Pointer);

            /* Last object? */
            if(Pointer == NULL)
                break;

            /* Access the type of the geometry as Type */
            CDType(Pointer,&Type);
```

```
                    /* Dispatch according to Type to specific procedure */
                    if(Type == CDBox) {
                         /* Access the box */
                         CDBox(Pointer,&Layer,...);
                         /* Transform the box's center. */
                         TPoint(&X,&Y);
                         /* Display the box. */
                         DisplayBox(X,Y,Length,Width);
                         }
                    else if(Type == CDWire){
                         /* Access the wire */
                         CDWire(Pointer,&Layer,...);
                              .
                              .
                              .
                         }
                       .
                       .
                      etc.
                       .
                       .
                    }
               }
          }
```

**THE CD.H FILE**

The following lists all relevant definitions contained within the *cd.h* file.

```
#include <stdio.h>
#include <math.h>
#include "nmalloc.h"                    /* new malloc routines */
#include "macros.h"                     /* user macro package */


#define   FILENAMESIZE        16        /* maximum size of a file name */


/*
 * Values routines return in StatusInt of CDOpen, CDBeginMakeCall,
 * CDTo, CDFrom, or CDParseCIF.
 */
#define   CDPARSEFAILED       1         /* (FATAL) parse failed */
#define   CDOLDSYMBOL         2         /* symbol already in database */
#define   CDNEWSYMBOL         3         /* symbol not in search path */
#define   CDSUCCEEDED         4         /* new symbol(s) found in path */
```

```
/*
 * Valid arguments to CDError().
 */
#define    CDMALLOCFAILED        11       /* (FATAL) out of memory */
#define    CDBADBOX              12       /* zero width or length box */
#define    CDXFORMSTACKFULL      13       /* transform stack overflow */
#define    CDBADPATH             14       /* bad directory name in search path */


/*
 * Types of geometries
 */
#define    CDSYMBOLCALL          'c'
#define    CDPOLYGON             'p'
#define    CDROUNDFLASH          'r'
#define    CDLABEL               'l'
#define    CDWIRE                'w'
#define    CDBOX                 'b'


/*
 * Types of transformations
 */
#define    CDMIRRORX             'x'      /* mirror in the direction of x */
#define    CDMIRRORY             'y'      /* mirror in the direction of y */
#define    CDROTATE              'r'      /* rotate by vector X,Y */
#define    CDTRANSLATE           't'      /* translate to X,Y */


/*
 * CD Control flags; See struct d below.
 */
#define    DCONTROLCDOPEN        'o'
#define    DCONTROLCDTO          't'
#define    DCONTROLVANILLA       'v'


/*
 * Coordinate system with 1 micron features and 1 cm dice.
 * Remember that there are 100 CIF units per micron.
 */
#define    INFINITY              100000000
```

```
/*
 * These are the numbers that CD uses to determine which bin an object
 * resides in. They should reflect the average size of a layout being
 * edited by KIC. KIC will not fail if the numbers are too small.
 * Anything outside of this window is placed in the residual bin.
 * If these numbers become too large, CDIntersect() must use floating
 * point calculations.
 */
#define   CDBINMAXX            500000
#define   CDBINMAXY            500000
#define   CDBINMINX            ( -CDBINMAXX )
#define   CDBINMINY            ( -CDBINMAXY )


/*
 * PLEASE NOTE
 * ^^^^^^^^^^^
 * Because a char is used as the layer and info fields, the absolute
 * maximum number of layers is 255. This may be increase by
 * recompiling kic with the Layer and Info fields typed as ints.
 */
#define   CDNUMBINS            12
#define   CDNUMLAYERS          35


/*
 * Number of symbols stored in the symbol table for any given cell.
 */
#define   CDNUMREMEMBER        1000


/*
 * Storage for diagnostics of CDError().
 */
extern char *CDStatusString;
extern int CDStatusInt;


/*
 * Master list desc.
 */
struct m {
     int mReferenceCount;
     int mLeft,mBottom,mRight,mTop;
     char *mName;
     struct m *mPred,*mSucc;
     };
```

```
/*
 * Property List desc.
 */
struct prpty {
        int prpty_Value;
        char *prpty_String;
        struct prpty *prpty_Succ;
        };


/*
 * Symbol desc.
 */
struct s {
        int sLeft,sBottom,sRight,sTop;
        int sBBValid;
        int sA,sB;
        char *sName;
        short sInfo;
        /*
         * One bin for each layer.  Layer 0 is for call descs.
         * Each bin points to a double linked list of object descs.
         * Bin[.][0][0] are the RESIDUAL bins–Bin[.][0][1] and Bin[.][1][0]
         * are unused.  NumBins should be as big as it can be.
         * For 20 layers and 100 bins per layer,
         * the data structure becomes 2000 words.
         */
        struct o ***sBin[CDNUMLAYERS+1];
        struct m *sMasterList;
        struct prpty *sPrptyList;
        };


/*
 * Object desc.
 */
struct o {
        int oLeft,oBottom,oRight,oTop;
        char oType;
        char oLayer;
        short oInfo;
        struct o *oPred,*oSucc;
        struct o *oRep;
        struct prpty *oPrptyList;
        };
```

```
/*
 * Polygon desc.
 */
struct po {
        struct p *poPath;
        };


/*
 * Round flash desc.
 */
struct r {
        int rWidth,rX,rY;
        };


/*
 * Wire desc.
 */
struct w {
        int wWidth;
        struct p *wPath;
        };


/*
 * Call desc.
 */
struct c {
        int cNum;
        /* Pointer to transformation descriptor. */
        struct t *cT;
        /* Pointer to master list descriptor. */
        struct m *cMaster;
        /* Array parameters. */
        int cNumX,cNumY,cDX,cDY;
        };


/*
 * Transform desc.
 * If MX, tType == CDMIRRORX.  If MY, tType == CDMIRRORX.
 * If R, tType == CDROTATE, tX == XDirection, tY == YDirection.
 * If T, tType == CDTRANSLATE, tX == TX, tY = TY;
 */
struct t {
        char tType;
        int tX,tY;
        struct t *tSucc;
        };
```

```
/*
 * Label desc.
 */
struct la {
        char *laLabel;
        int laX,laY;
        };


/*
 * Linked path structure
 */
struct p {
        int pX,pY;
        struct p *pSucc;
        };


/*
 * Generator desc.
 * Search Bin[Layer][0][0] first.
 * Then Bin[Layer][BeginX..EndX][BeginY..EndY].
 * Bin[Layer][X][Y] is the current bin.
 * Pointer points to the current desc in the current bin.
 */
struct g {
        int  gLeft,gBottom,gRight,gTop;
        int  gBeginX,gX,gEndX,gBeginY,gY,gEndY;
        char gLayer;
        struct o *gPointer;
        };


/*
 * CD's current parameter struct
 */
struct d {
        /*
         * DCONTROLCDOPEN denotes CD is in CDOpen
         * DCONTROLCDTO denotes CD is in CDTo
         * DCONTROLVANILLA denotes CD is in none of the above
         */
        char dControl;

        /* Current parameters for symbol being parsed in CDOpen. */
        int  dNumX,dDX,dNumY,dDY;

        /* Scale factors for CDTo and CDFrom. */
        int dA,dB;

        /* Symbol scale factors. */
        int dDSA,dDSB;
```

```
struct o *dPointer;
struct s *dSymbolDesc;
struct s *dRootCellDesc;

/* UNIX file names are limited to 14 characters */
char dSymbolName[FILENAMESIZE];
FILE *dSymbolFileDesc;

/*
 * Fields used in CDTo follow.
 */

/* True if parsing root symbol. */
int dRoot;

/* Root's file desc. */
FILE *dRootFileDesc;

/* Current property list for symbol being parsed */
struct prpty *dPrptyList;

/*
 *   dProgram  ==  'h' if IGS gened it.
 *   dProgram  ==  'i' if Icarus gened it.
 *   dProgram  ==  's' if Sif gened it.
 *   dProgram  ==  'n' if none of the above.
 */
char dProgram;

/*
 * Symbol name table.
 * 16 comes from the fact that a UNIX file name is only 14 characters
 * long and each symbol name is a UNIX file name.  VMS file names are
 * smaller.
 */
char dSymTabNames[CDNUMREMEMBER][FILENAMESIZE];
int dSymTabNumbers[CDNUMREMEMBER];
int dNumSymbolTable;

/*
 * Because CIF files may have FORWARD references, CDTo must pass
 * over the CIF file TWICE.
 * On the first pass, it just fills up the symbol name table.
 * On the second pass, it does the translation to KIC format.
 */
int dFirstPass;
```

```
          /*
           * True if debugging.
           */
          int dDebug;
          int dNumSymbolsAllocated;
          }
      CDDesc;


      /*
       * CD layer table
       */
      struct l {
           char lTechnology;
           char lMask[3];
           /*True if CDFrom should output layer.*/
           char lCDFrom;
           }
      CDLayer[CDNUMLAYERS+1];


      /*
       * Hash table of symbol descs keyed on symbol's name.
       */
      struct bu {
           struct s *buSymbolDesc;
           struct bu *buPred;
           struct bu *buSucc;
           };
      struct bu *CDSymbolTable[CDNUMLAYERS];
```

**BUGS**

Labels can cause difficulty to an application program because *CD* has no notion of font size. Therefore, the bounding box of a label is the lower, left coordinate to which the label is justified.

**FILES**

~cad/src/kic/cd.c
~cad/src/kic/cd.h
~cad/src/kic/actions.c
~cad/src/kic/parser.c
~cad/src/kic/parser.h
~cad/src/kic/paths.c
~cad/src/kic/xforms.c
~cad/src/kic/xforms.h

**SEE ALSO**

kic(CAD1)

# Appendix C

# The KIC Tutorial User's Guide

The KIC tutorial user's guide is contained in this appendix. This user's guide is appropriate for KIC running under both UNIX and VMS.

The tutorial is supplied separately as part of the program on tape, as it changes periodically.

# Appendix D

# The MFB Programmers's Manual

The Section 3 UNIX manual pages for the Model Frame Buffer graphics package are contained in this appendix.

## NAME

mfb – model frame buffer interface

## SYNOPSIS

**#include <~cad/include/mfb.h>** in the program source.
**cc** [ flags ] files **~cad/lib/mfb.a -lm** [ libraries ]

## DESCRIPTION

These routines provide the user with a virtual graphics interface. They perform the terminal dependent task of encoding/decoding graphics code, thereby allowing the user to write graphics programs to run on almost any graphics device.

The user opens and initializes a graphics device by calling the *MFBOpen* routine that returns a pointer to that device's *MFB* data structure defined at the end of this manual. By maintaining several *MFB* data structures, an application program can drive several graphics devices simultaneously. Once opened, *MFB* graphics routines can be called to draw geometries, draw graphics text, set device parameters, or receive keyboard input. An application program can also use any of the several utility routines that perform line clipping, polygon clipping, or window/viewport transformations. Control of the graphics device is released by calling the *MFBClose* routine.

All programs that use *MFB* routines must include the file *mfb.h* that defines the *MFB* data structure to contain the information provided by *MFBCAP(5).*

## INITIALIZATION ROUTINES

**MFB *MFBOpen(TerminalName, DeviceName, errorcode)**
**char *TerminalName, *DeviceName;**
**int *errorcode;**

*MFBOpen* initializes the graphics device and fills the *MFB* data structure with information found in *MFBCAP(5). TerminalName* is a pointer to a null terminated string containing the name of the graphics device as defined in the *mfbcap* database file. This argument has no default and can never be null. *DeviceName* is a pointer to a null terminated string containing the full path name to the respective graphics device. If null, *stdin* and *stdout* are used by default. *errorcode* is a diagnostic integer returned by *MFBOpen*. The possible returned values for *errorcode* are defined in the *mfb.h* file as follows:

```
#define   MFBOK          1     /* successful return */
#define   MFBBADENT    -10     /* Unknown terminal type */
#define   MFBBADMCF    -20     /* Can't open MFBCAP file */
#define   MFBMCELNG    -30     /* MFBCAP entry too long */
#define   MFBBADMCE    -40     /* Bad MFBCAP entry */
#define   MFBINFMCE    -50     /* infinite loop in MFBCAP entry */
#define   MFBBADTTY    -60     /* stdout not in /dev */
#define   MFBBADDEV   -180     /* Can't open or close device */
#define   MFBBADOPT   -190     /* Can't access or set device stat */
#define   MFBBADWRT   -220     /* Error during write */
```

Only MFBOK is not a fatal error.

**void SetCurrentMFB(mfb)**
**MFB *mfb;**

*SetCurrentMFB* allows the application program to define the current graphics device. All subsequent calls to *MFB* routines will affect the specified device. Because each *MFB* data structure contains a separate output buffer, it is not necessary to flush the output before resetting the current output device. *MFBOpen* returns with the opened graphics device defined as the current output device.

**int MFBInitialize()**

*MFBInitialize* will flush the output buffer and (re)initialize the device for graphics input. The graphics device or standard input will be placed in CBREAK mode. See the manual *tty(4)*. MFBOK is returned if the device was successfully initialized; MFBBADOPT is returned if an error was encountered while attempting to access or set the device status, and MFBBADTTY is returned if standard output can bot be found or accessed.

**int MFBClose()**

*MFBClose* will flush the output buffer and release control of the graphics device driver. If the graphics device is a *tty*, it is returned to the state that existed prior to the respective *MFBOpen* call. MFBOK is returned if the device was successfully closed; MFBBADOPT is returned if an error was encountered while attempting to access or set the device status, and MFBBADDEV is returned if the output device could not be closed.

**int MFBHalt()**

*MFBHalt* will flush the output buffer and release control of the graphics device driver. If the graphics device is a *tty*, it is returned to the state that existed prior to the respective *MFBOpen* call. *MFBHalt* differs from *MFBClose* in that the memory occupied by the respective, current *MFB* data structure is not freed. By calling *MFBInitialize*, the graphics device will be reinitialized. This routine is typically used by an application program for handling the SIGTSTP signal (the keyboard stop signal, usually control-Z ). MFBOK is returned if the device was successfully returned to its initial state; MFBBADOPT is returned if an error was encountered while attempting to access or set the device status.

## SETTING DEVICE PARAMETERS

Each of the following routines for setting device parameters returns a diagnostice integer that is defined in the *mfb.h* file as follows:

```
#define   MFBOK          1      /* successful return */
#define   MFBBADLST      -70    /* Illegal line style */
#define   MFBBADFST      -80    /* Illegal fill style */
#define   MFBBADCST      -90    /* Illegal color style */
#define   MFBBADTM1      -100   /* No destructive text mode */
#define   MFBBADTM2      -110   /* No overstriking text mode */
#define   MFBNOBLNK      -150   /* No definable blinkers */
#define   MFBTMBLNK      -160   /* Too many blinkers */
#define   MFBNOMASK      -170   /* No definable read or write mask */
#define   MFBBADALU      -250   /* Cannot set ALU mode */
```

**int MFBSetLineStyle(styleId)**
**int styleId;**

> *MFBSetLineStyle* sets the current line style to that identified by the integer *styleId* that is greater than or equal to zero and less than the value of *maxLineStyles* in the *MFB* data structure. The value of *maxLineStyles* can be obtained from the *MFBInfo* routine defined below. Zero is always the *styleId* for solid lines. Except for the solid line style, *MFB* assumes no default set of lines styles. MFBOK is returned if the line style was successfully set to that specified by *styleId* or if *styleId* was already the current line style; MFBBADLST is returned if *styleId* has an illegal value.

**int MFBSetFillPattern(styleId)**
**int styleId;**

> *MFBSetFillPattern* sets the current fill pattern to that identified by the integer *styleId* that is greater than or equal to zero and less than the value of *maxFillPatterns* in the *MFB* data structure. The value of *maxFillPatterns* can be obtained from the *MFBInfo* routine defined below. Solid fill is always defined by *styleId* equal to zero. Other than solid fill, *MFB* assumes no default set of fill patterns. MFBOK is returned if the fill style was successfully set to that specified by *styleId* or if *styleId* was already the current fill style; MFBBADFST is returned if *styleId* has an illegal value.

**int MFBSetChannelMask(channelMask)**
**int channelMask;**

> *MFBSetChannelMask* defines the current write mask to be the value of *channelMask*. The channel mask allows specific memory planes to be written and erased without disturbing other memory planes. The least significant bit of *channelMask* corresponds to the masked value of the first memory plane, etc. If the corresponding bit is zero, the memory plane is write-protected. The number of memory planes can be obtained from the *MFBInfo* routine defined below. MFBOK is returned if the write mask was successfully set to *channelMask* or if *channelMask* was already the current write mask; MFBNOMASK is returned if the graphics device does not have a definable write mask.

**int MFBSetReadMask(readmask)**
**int readmask;**

> *MFBSetReadMask* defines the current read mask to be the value of *readmask*. The read mask allows only specific memory planes to be read. MFBOK is returned if the read mask was successfully set to *readMask* or if *readMask* was already the current read mask; MFBNOMASK is returned if the graphics device does not have a definable read mask.

**int MFBSetColor(colorId)**
**int colorId;**

*MFBSetColor* sets the current foreground color to that identified by the integer *colorId* that is greater than or equal to zero and less than the value of *maxColors* in the *MFB* data structure. The value of *maxColors* can be obtained from the *MFBInfo* routine defined below. There is no default color map in *MFB*. MFBOK is returned if the foreground color was successfully set to that specified by *colorId* or if *colorId* was already the current foreground color; MFBBADCST is returned if *colorId* has an illegal value.

**int MFBSetTextMode(destructiveBool)**
**Bool destructiveBool;**

*MFBSetTextMode* defines whether subsequent graphics text will overstrike or replace previous text. If *destructiveBool* is true, the text mode is set to destructive which means that graphic text will set the background color of the font grid to the color that is specified by color style zero depending on the currently defined ALU operation. Overstriking mode will only set the pixels of the character font to the current foreground color. MFBOK is returned if the graphic text writing mode was successfully set to that specified by *destructiveBool* or if *destructiveBool* was already the current graphic text writing mode; MFBBADTM1 is returned if the graphics device does not have a destructive graphic text mode, and MFBBADTM2 is returned if the graphics device does not have an overstriking graphic text mode.

**int MFBSetALUMode(alumode)**
**int alumode;**

*MFBSetALUMode* changes the mode by which the graphics display is changed when an area of the display is over-written. The four possible modes are JAM (replace mode), OR, EOR (exclusive OR), and NOR. The four valid arguments to *MFBSetALUMode* are defined in *mfb.h* as follows:

```
#define   MFBALUJAM      0      /* set ALU mode to JAM */
#define   MFBALUOR       1      /* set ALU mode to OR */
#define   MFBALUNOR      2      /* set ALU mode to NOR */
#define   MFBALUEOR      3      /* set ALU mode to EOR */
```

MFBOK is returned if the ALU mode was successfully set to that specified by *alumode* or if *alumode* was already the current ALU operation; MFBBADALU is returned if the graphics device does not have the ALU mode specified by *alumode* or if *alumode* is an invalid or illegal argument.

**int MFBSetCursorColor(colorId1, colorId2)**
**int colorId1, colorId2;**

*MFBSetCursorColor* sets the graphics cursor to blink between the two colors identified by *colorId1* and *colorId2*. The constraints on the values for *colorId1* and *colorId2* are the same as for *MFBSetColor* defined above. The frequency of the blinking cursor is fixed and can be changed only by modifying the *mfbcap* database file. MFBOK is returned if the blinking cursor colors were successfully set to the specified colors.

**int MFBSetRubberBanding(onFlag, X, Y)**
**int X, Y;**
**Bool onFlag;**

*MFBSetRubberBanding* enables/disables rubber banding of the pointing device. If *onFlag* is false, then rubber banding is disabled. When enabled, the center of rubber banding is at *X, Y.* Rubber banding is always disabled immediately after the pointing device has been used. MFBOK is returned if the rubberbanding mode was successfully set to that specified by *onFlag;* MFBNORBND is returned if the graphics device does not have rubberbanding in the pointing device.

**int MFBSetBlinker(colorId, red, green, blue, onFlag)**
**int colorId;**
**int red, green, blue;**
**int onFlag;**

*MFBSetBlinker* enables the color identified by *colorId* to blink between its currently defined color and the color defined by the *red, green, blue* combination. The values of *red, green,* and *blue* are normalized to 1000. If *onFlag* is zero, the blinking is disabled. The number of colors that may be defined as blinkers at any given time must be less than the value of *maxBlinkers* in the *MFB* data structure. The frequency of the blinking colors is fixed and can be changed only by modifying the *mfbcap* database file. MFBOK is returned if the color specified by *colorId* was successfully set to the desired blinking mode; MFBNOBLNK is returned if the graphics device does not have blinking VLT layers, and MFBTMBLNK is returned if there are already too many active blinking layers.

## DEFINING DEVICE PARAMETERS

Each of the following routines for defining device parameters returns a negative value if any difficulty is encountered. The possible returned integers are defined in *mfb.h* as follows:

```
#define   MFBOK           1     /* successful return */
#define   MFBNODFLP    -120     /* No definable line patterns */
#define   MFBNODFFP    -130     /* No definable fill patterns */
#define   MFBNODFCO    -140     /* No definable colors */
```

**int MFBDefineColor(colorId, red, green, blue)**
**int colorId;**
**int red, green, blue;**

*MFBDefineColor* redefines the VLT entry for the color identified by *colorId* to be the color represented by the *red, green, blue* combination where *red, green,* and *blue* are normalized to 1000. Once the color corresponding to *colorId* is redefined, all geometries that were written onto the display of a frame buffer with *colorId* as the current color will become the new color. MFBOK is returned if the VLT entry for *colorId* was successfully defined; MFBNODFCO is returned if the graphics device does not have a VLT.

**int MFBDefineFillPattern(styleId, BitArray)**
**int styleId;**
**int *BitArray;**

*MFBDefineFillPattern* redefines the fill pattern identified by ***styleId*** and returns with ***styleId*** as the current fill style. *BitArray* is a pointer to an array of eight integers whose least significant eight bits represent individual rows in an eight by eight intensity array. For example, a fill pattern with an ascending diagonal line may be defined by the following eight (decimal) integers:

1 2 4 8 16 32 64 128 256

A diagonal-grid fill pattern can be defined with the following integer array:

257 130 68 40 40 68 130 257

MFBOK is returned if the new fill style for ***styleId*** was successfully defined; MFBNODFFP is returned if the graphics device does not have definable fill patterns.

**int MFBDefineLineStyle(styleId, Mask)**
**int styleId;**
**int Mask;**

*MFBDefineLineStyle* defines the line style identified by ***styleId*** to be the pattern contained in the eight least significant bits of ***Mask*** and returns with ***styleId*** as the current line style. MFBOK is returned if the line style for ***styleId*** was successfully defined; MFBNODFFP is returned if the graphics device does not have definable line patterns.

## INPUT/OUTPUT ROUTINES

**int MFBUpdate()**

*MFBUpdate* flushes the internal output buffer to the currently defined output device and will ignore any write error that may occur. A call to this routine is ABSOLUTELY necessary to complete any graphics display sequence. *MFBUpdate* returns the number of characters sent to the output graphics device or -1 if a write error occured.

**int MFBPoint(X, Y, key, button)**
**int *X, *Y, *button;**
**char *key;**

*MFBPoint* enables the graphics pointing device and then waits for user input. If a keyboard key is pressed, *MFBPoint* returns with ***key*** containing the character that was pressed. If the pointing device is pressed, *MFBPoint* returns with the identified viewport coordinate *X*, *Y*, the contents of ***key*** equal to zero, and the button mask of the button that was pushed. The integer array ***buttonMask*** in the *MFB* data structure contains all possible button mask values that can be returned. *MFBPoint* returns one of the following values defined in the ***mfb.h*** file:

```
#define   MFBOK          1     /* successful return */
#define   MFBPNTERR   -230     /* Error in pointing device */
#define   MFBNOPTFT   -240     /* No pointing format */
#define   MFBNOPNT    -260     /* No pointing device */
```

**char *MFBKeyboard(X, Y, background, foreground)**
**int background;**
**int foreground;**
**int X, Y;**

*MFBKeyboard* enables the graphics keyboard and waits for user input. A pointer to a character buffer containing the keyboard input is returned by *MFBKeyboard* when the user presses the return or linefeed key. Backspace is control-H or the delete character, and control-X or control-U will kill the line. Pressing the ESCAPE key will cause *MFBKeyboard* to return with a null string in the input character buffer.

All keyboard input is displayed in the viewport with the lower left corner at the viewport coordinate *X*, *Y* and is constrained to fit on one line. *Background* and *foreground* are the background and foreground color styles respectively in which the keyboard input will be displayed.

**void MFBAudio()**

*MFBAudio* will ring the bell or alarm on the graphics device. If the graphics device does not have a bell, then a control-G will be sent to standard output.

**int MFBPutchar(c)**
**char c;**

**int MFBPutstr(cp,nchars)**
**int nchars;**
**char *cp;**

**int MFBGetchar()**

These three routines are used internally by *MFB* and typically are not used within an application program. They are comparable to the *stdio* routines having similar names. *MFBPutchar* places a character *c* in the output buffer. *MFBPutstr* inserts a string pointed to by *cp* containing *nchars* characters into the output buffer. The *nchars* argument is necessary to permit embedded null characters in the output stream. The characters remain in the output buffer until the next call to *MFBUpdate* or until the contents of the output buffer exceed 4096 characters.

*MFBGetchar* returns a single character from the graphics input device. If the graphics device does not have a keyboard, input is obtained from the terminal from which the application program was invoked. If the graphics device is a *tty*, it should be remembered that it is in CBREAK mode.

## TWO DIMENSIONAL GEOMETRY ROUTINES

All coordinates that are passed to the following geometry routines are with respect to the display resolution of the graphics device. *MFB* assumes that the lower, left corner of the display is the origin with an absolute coordinate (0, 0). All coordinate values are positive integers.

**void MFBMoveTo(X1, Y1)**
**int X1, Y1;**

*MFBMoveTo* sets the current graphics position to *X1, Y1*. No line will be drawn from the old graphics position.

**void MFBDrawLineTo(X1, Y1)**
**int X1, Y1;**

*MFBDrawLineTo* draws a line from the current graphics position to *X1, Y1* in the current line style and color. The current graphics position then becomes *X1, Y1*.

**void MFBLine(X1, Y1, X2, Y2)**
**int X1, Y1, X2, Y2;**

*MFBLine* draws a line in the current line style and color from *X1, Y1* to *X2, Y2*.

**void MFBBox(left, bottom, right, top)**
**int left, bottom, right, top;**

*MFBBox* displays a rectangle in the current fill pattern and color with diagonal coordinates at *left, bottom* and *right, top*.

**void MFBDrawPath(path)**
**MFBPATH *path;**

*MFBDrawPath* draws a path of vectors in the current line style and color. *Path* is a pointer to a data structure defined in the *mfb.h* file as follows:

```
struct mfbpath {
        int nvertices;        /* number of (x,y) coordinate pairs */
        int *xy;              /* pointer to array of (x,y) coordinates */
        };

typedef struct mfbpath MFBPATH;
```

The contents of the coordinate array are organized such that xy[0] is the x coordinate of the first vertex, xy[1] is the y coordinate of the first vertex, xy[2] is the x coordinate of the second vertex, etc.

**void MFBFlood()**

*MFBFlood* erases a frame buffer display to the current color as previously defined by *MFBSetColor*. The result would be the same as drawing a solid box over the entire display.

**void MFBPixel(X, Y)**
**int X, Y;**

*MFBPixel* sets the pixel at location *X, Y* on the display to the current color as previously defined by *MFBSetColor*.

**void MFBCircle(X, Y, rad, nsides)**
**int X, Y, rad, nsides;**

*MFBCircle* draws the perimeter of a circle in the current line style and color with center at *X, Y* and with radius *rad*. The argument *nsides* is the number of line segments with which the circle will be drawn if the frame buffer does not have a circle primitive. The default value for *nsides* is twenty.

**void MFBFlash(X, Y, rad, nsides)**
**int X, Y, rad, nsides;**

*MFBFlash* draws a round flash with the current fill pattern and color with center at *X, Y* and radius *rad*. The argument *nsides* is the number of line segments with which the flash will be drawn. The default value for *nsides* is twenty.

**void MFBArc(X, Y, rad, angle1, angle2, nsides)**
**int X, Y, rad;**
**int angle1, angle2, nsides;**

*MFBArc* draws an arc in the current line style and color with center at *X, Y* and with radius *rad* beginning at *angle1* with respect to the positive y–axis and ending at *angle2* in a counter-clockwise direction. Both angles are in degrees and are greater than or equal to zero and less than or equal to 360. The argument *nsides* is the number of line segments with which a 360 degree arc will be drawn. The default value for *nsides* is twenty.

**void MFBPolygon(poly)**
**MFBPOLYGON *poly;**

*MFBPolygon* draws a polygon with the current fill pattern and color. *Poly* is a pointer to a data structure defined in the *mfb.h* file as follows:

```
struct mfbpolygon {
        int nvertices;        /* number of (x,y) coordinate pairs */
        int *xy;              /* pointer to array of (x,y) coordinates */
        };

typedef struct mfbpolygon MFBPOLYGON;
```

The contents of the coordinate array are organized such that xy[0] is the x coordinate of the first vertex, xy[1] is the y coordinate of the first vertex, xy[2] is the x coordinate of the second vertex, etc. The difference between the *MFBPOLYGON* typedef and the *MFBPATH* typedef defined above is that the *MFBPOLYGON* struct is assumed to define a closed path of coordinates.

**MFBPATH *MFBArcPath(X, Y, rad, angle1, angle2, nsides)**
**int X, Y, rad;**
**int angle1, angle2, nsides;**

*MFBArcPath* returns a pointer to a *MFBPATH* struct that contains an arc with center at *X, Y* and with radius *rad* beginning at *angle1* with respect to the positive y–axis and ending at *angle2* in a counter-clockwise direction. Both angles are in degrees and are greater than or equal to zero and less than or equal to 360. The argument *nsides* is the number of line segments with which the arc will be drawn. The default value for *nsides* is twenty.

**MFBPOLYGON *MFBEllipse(X, Y, radx, rady, nsides)**
**int X, Y, rad, nsides;**

*MFBEllipsePath* returns a pointer to a *MFBPOLYGON* struct that contains an elliptical polygon with center at *X, Y* and with distance *radx* from the center to an edge along the x–axis and distance *rady* from the center to an edge along the y–axis. The argument *nsides* is the number of line segments with which the arc will be drawn. The default value for *nsides* is twenty.

**void MFBText(text, X, Y, phi)**
**char *text;**
**int X, Y, phi;**

*MFBText* displays a null terminated string pointed to by *text* with the lower left corner at *X, Y* in the display viewport with the current color and rotated at the angle *phi* in degrees. The default value for *phi* is zero.

**void MFBNaiveBoxFill(left, bottom, right, top)**
**int top, bottom, left, right;**

*MFBNaiveBoxFill* can be used to draw a filled rectangle on a graphics device that does not support fill patterns. The *MFBLine* routine is used to draw rectangles with eight different fixed fill styles. If the graphics device does not have a command primitive for drawing a box, then the *MFBBox* routine defined above defaults to *MFBNaiveBoxFill.*

## RASTER ROUTINES

**void MFBRasterCopy(X,Y,DX,DY,DestX,DestY);**
**int X,Y,DX,DY,DestX,DestY;**

*MFBRasterCopy* copies a rectangular area with the bottom, left corner at *X,Y* and with length *DX* in the X direcion and width *DY* in the Y direction to an area with the bottom, left corner at *DestX,DestY*. The mode of the copy operation was specified by the last call to *MFBSetALUMode*.

## INFORMATION ACQUISITION

**int MFBInfo(info);**
**int info;**

*MFBInfo* is a routine for obtaining device specific information. The possible values for *info* are defined in *mfb.h* as follows:

```
#define   MAXX                 1     /* max x coordinate */
#define   MAXY                 2     /* max y coordinate */
#define   MAXCOLORS            3     /* max number of colors */
#define   MAXINTENSITY         4     /* max color intensity */
#define   MAXFILLPATTERNS      5     /* max number of fill patterns */
#define   MAXLINESTYLES        6     /* max number of line styles */
#define   MAXBLINKERS          7     /* max number of blinkers */
#define   POINTINGDEVICE       8     /* terminal has pointing device */
#define   POINTINGBUTTONS      9     /* pointing device has buttons */
```

```
#define   NUMBUTTONS          10     /* num. of pointing dev. buttons */
#define   BUTTON1             11     /* value returned by button 1 */
#define   BUTTON2             12     /* value returned by button 2 */
#define   BUTTON3             13     /* value returned by button 3 */
#define   BUTTON4             14     /* value returned by button 4 */
#define   BUTTON5             15     /* value returned by button 5 */
#define   BUTTON6             16     /* value returned by button 6 */
#define   BUTTON7             17     /* value returned by button 7 */
#define   BUTTON8             18     /* value returned by button 8 */
#define   BUTTON9             19     /* value returned by button 9 */
#define   BUTTON10            20     /* value returned by button 10 */
#define   BUTTON11            21     /* value returned by button 11 */
#define   BUTTON12            22     /* value returned by button 12 */
#define   TEXTPOSITIONALBE    30     /* Bool: positionable text */
#define   TEXTROTATABLE       31     /* Bool: rotatable text */
#define   FONTHEIGHT          32     /* font height in pixels */
#define   FONTWIDTH           33     /* font width in pixels */
#define   FONTXOFFSET         34     /* font x offset in pixels */
#define   FONTYOFFSET         35     /* font y offset in pixels */
#define   DESTRUCTIVETEXT     36     /* Bool: text can be destructive */
#define   OVERSTRIKETEXT      37     /* Bool: text can be overstrike */
#define   VLT                 38     /* Bool: terminal has VLT */
#define   BLINKERS            39     /* Bool: terminal has blinkers */
#define   FILLEDPOLYGONS      40     /* Bool: terminal can fill polygons */
#define   DEFFILLPATTERNS     41     /* Bool: definable fill patterns */
#define   DEFCHANNELMASK      42     /* Bool: definable write mask */
#define   DEFLINEPATTERN      43     /* Bool: definable line styles */
#define   CURFGCOLOR          44     /* current foreground color */
#define   CURFILLPATTERN      45     /* current fill pattern */
#define   CURLINESTYLE        46     /* current line style */
#define   CURCHANNELMASK      47     /* current channel mask */
#define   CURREADMASK         48     /* current read mask */
#define   NUMBITPLANES        49     /* number of bit planes */
#define   DEFREADMASK         50     /* Bool: definable read mask */
#define   RASTERCOPY          51     /* Bool: term has raster copy */
#define   OFFSCREENX          52     /* left of off screen memory */
#define   OFFSCREENY          53     /* bottom of off screen memory */
#define   OFFSCREENDX         54     /* length of off screen memory */
#define   OFFSCREENDY         55     /* width of off screen memory */
```

If an invalid argument is used, *MFBInfo* will return -1.

## WINDOW/VIEWPORT TRANSFORMATIONS

*MFB* provides a set of procedures for converting from window coordinates to viewport coordinates and vice versa. These transformation routines are NOT used by the *MFB* display routines and must be invoked separately by an application program.

**void MFBSetViewport(left, bottom, right, top)**
**int left, bottom, right, top;**

**void MFBSetWindow(left, bottom, right, top)**
**int left, bottom, right, top;**

**int MFBScaleX(X)**  
**int X;**

**int MFBScaleY(Y)**  
**int Y;**

**int MFBDescaleX(X)**  
**int X;**

**int MFBDescaleY(Y)**  
**int Y;**

To use these routines, it is necessary to define both the viewport of the graphics display and the window in the working area by using *MFBSetViewport* and *MFBSetWindow*. The viewport must always be defined by non-negative integers that are within the resolution of the graphics display. Once defined, *MFBScaleX* and *MFBScaleY* will convert from window coordinate values to viewport coordinates. *MFBDescaleX* and *MFBDescaleY* will perform the inverse transformation. The transform routines return the scaled values.

## GEOMETRY CLIPPING ROUTINES

*MFB* provides a set of routines for clipping lines and polygons to a given window.

**void MFB_Y_Intercept(X1, Y1, X2, Y2, value, Yvalue)**  
**int X1, Y1, X2, Y2;**  
**int value;**  
**int *Yvalue;**

*MFB_Y_Intercept* calculates the value *Yvalue* of the y coordinate at the point of intersection of a line defined by the two coordinates *X1, Y1* and *X2, Y2*, and a vertical line with all x coordinates equal to *value*.

**void MFB_X_Intercept(X1, Y1, X2, Y2, value, Xvalue)**  
**int X1, Y1, X2, Y2;**  
**int value;**  
**int *Xvalue;**

*MFB_X_Intercept* calculates the value *Xvalue* of the x coordinate at the point of intersection of a line defined by *X1, Y1* and *X2, Y2*, and a horizontal line with all y coordinates equal to *value*.

**void MFBLineClip(X1, Y1, X2, Y2, left, bottom, right, top);**  
**int *X1, *Y1, *X2, *Y2;**  
**int left, bottom, right, top;**

The above clipping routines are used by *MFBLineClip* to clip the line segment defined by *X1, Y1* and *X2, Y2* to the window defined by *left*, *bottom*, *right*, and *top*.

**void MFBPolygonClip(poly, top, bottom, left, right)**  
**MFBPOLYGON *poly;**  
**int top, bottom, left, right;**

*MFBPolygonClip* clips a polygon with less than 200 vertices defined by *poly* to the window defined by *left*, *bottom*, *right*, and *top*. *poly* is replaced by the clipped polygon.

**MFBPATH *MFBArcClip(path, left, bottom, right, top)**
**MFBPATH *path;**
**int left, bottom, right, top;**

*MFBArcClip* clips an arc with less than 200 vertices defined by *path* to the window defined by *left*, *bottom*, *right*, and *top*. *MFBClipArc* returns a pointer to an array of five *MFBPATH* structs that define the clipped arc. The contents of several of these returned structs may define a null path.

## SPECIAL VIEWPORT ROUTINES

**void MFBMore(left, bottom, right, top, Textfile)**
**int left, bottom, right, top;**
**FILE *Textfile;**

*MFBMore* will display the contents of a file *Textfile* in a viewport defined by *left*, *bottom*, *right*, and *top* in a manner similar to the UCB program *MORE(1)*.

**void MFBScroll(left, bottom, right, top, Textfile)**
**int left, bottom, right, top;**
**FILE *Textfile;**

*MFBScroll* is an enhanced version of *MFBMore* defined above that allows you to scroll up or down through the contents of *Textfile*. The contents of *Textfile* are displayed in a viewport defined by *left*, *bottom*, *right*, and *top*.

## DIAGNOSTICS

**char *MFBError(errnum)**
**int errnum;**

*MFBError* can be used to obtain a null terminated string that describes the error associated with any of the above mentioned error codes. *errnum* is the error code returned by a *MFB* routine. *MFBError* returns a pointer to the error information string. The possible values for *errnum* are defined in *mfb.h* as follows:

```
#define   MFBBADENT     -10    /* Unknown terminal type */
#define   MFBBADMCF     -20    /* Can't open mfbcap file */
#define   MFBMCELNG     -30    /* MFBCAP entry too long */
#define   MFBBADMCE     -40    /* Bad mfbcap entry */
#define   MFBINFMCE     -50    /* Infinite mfbcap entry */
#define   MFBBADTTY     -60    /* stdout not in /dev */
#define   MFBBADLST     -70    /* Illegal line style */
#define   MFBBADFST     -80    /* Illegal fill style */
#define   MFBBADCST     -90    /* Illegal color style */
#define   MFBBADTM1     -100   /* No destructive text */
#define   MFBBADTM2     -110   /* No overstriking text */
#define   MFBNODFLP     -120   /* No definable line styles */
#define   MFBNODFFP     -130   /* No definable fill styles */
#define   MFBNODFCO     -140   /* No definable colors */
#define   MFBNOBLNK     -150   /* No blinkers */
#define   MFBTMBLNK     -160   /* Too many blinkers */
#define   MFBBADDEV     -180   /* Can't open or close device */
#define   MFBBADOPT     -190   /* Can't access or set device stat */
#define   MFBNOMASK     -170   /* No definable read or write mask */
```

```
#define    MFBBADWRT     -200    /* Error in write */
#define    MFBPNTERR     -210    /* Error in pointing device */
#define    MFBNOPTFT     -220    /* No format for pointing device */
#define    MFBNOPNT      -230    /* No pointing device */
#define    MFBNORBND     -240    /* No Rubberbanding */
#define    MFBBADALU     -250    /* Cannot set ALU mode */
```

**void MFBZeroCounters()**

**void MFBCounters(nCh,nBx,BxArea,nLn,LnLngth,nPxl)**
**int *nCh,*nBx,*BxArea,*nLn,*LnLngth,*nPxl;**

*MFBCounters* provides the ability to measure the communications bandwidth between the host and the graphics device. To use these routines, MFB must be compiled with the **DEBUG** flag defined in the ***mfb.h*** file.

The procedure is initialized by invoking *MFBZeroCounters* after which the geometry display routines such as *MFBBox* or *MFBLine* may be called in any order. When *MFBCounters* is invoked, it will return the number of graphic text characters *nCh* that where displayed, the number of boxes *nBx* displayed, the number of lines *nLn* displayed, the average pixel area of a box *BxArea*, the average line length *LnLngth* in pixels, and the total number of pixels *nPxl* that where affected. Invocation of *MFBCounters* does not clear the the counters.

## THE MFB DATA STRUCTURE

The *MFB* data structure is listed below.

```
#define   TTY      't'
#define   HCOPY    'r'

typedef enum {false, true} Bool;

struct mfb_window {
      int left;
      int right;
      int top;
      int bottom;
      double length,width;
      };

typedef struct mfb_window WINDOW;
typedef struct mfb_window VIEWPORT;

struct mfbpath {
      int nvertices;
      int *xy;
      };

typedef struct mfbpath MFBPOLYGON;
typedef struct mfbpath MFBPATH;
```

```
#ifndef vms
struct mfbremttyb {
        struct sgttyb oldttyb;
        struct sgttyb newttyb;
        };

struct mfbremstat {
        int graphttyw;                  /* old mode bits of graphics device */
        int kybrdttyw;                  /* old mode bits of standard I/O */
        struct stat graphstat;          /* old fstats of graphics device */
        struct stat kybrdstat;          /* old fstats of standard I/O */
        };

typedef struct mfbremttyb MFBSAVETTYB;
typedef struct mfbremstat MFBSAVESTAT;
#endif

struct mfbformatstrs {
        char *startSequence;            /* first transmitted sequence */
        char *endSequence;              /* last transmitted sequence */
        char *initLineStyles;           /* initialize line styles */
        char *initFillPatterns;         /* initialize fill styles */
        char *initColorStyles;          /* initialize color styles */

        char *vltEntry;                 /* define color in VLT */
        char *setForegroundColor;       /* set current foreground color */
        char *screenFlood;              /* flood screen to current color */

        char *channelMaskSet;           /* set write mask */
        char *readMaskSet;              /* set read mask */

        char *enablePointingDevice;     /* initialize pointing device */
        char *enableRubberBanding;      /* turn on rubber banding */
        char *disablePointingDevice;    /* disable pointing device and cursor */
        char *disableRubberBanding;     /* turn off rubber banding */
        char *readPointingDevice;       /* wait and read pointing device */
        char *formatPointingDevice;     /* decode format for pointing device */

        char *keyboardStart;            /* initailize keyboard */
        char *keyboardEnd;              /* terminate keyboard input */
        char *keyboardBackspace;        /* keyboard backspace sequence */
        char *audio;                    /* ring the terminals bell */
```

```
	char *lineDefineStart;           /* begin defining a line pattern */
	char *lineDefineFormat;          /* define bit array of line pattern */
	char *lineDefineEnd;             /* terminate line pattern definition */
	char *setLineStyle;              /* set current line style */
	char *setSolidLineStyle;         /* set current line style to solid */
	char *movePenSequence;           /* move current graphics position */
	char *drawLineSequence;          /* draw a line in current style */
	char *drawSolidLineSequence;     /* draw a solid line */
	char *drawLineToSequence;        /* move and draw current position */
	char *drawSolidLineToSequence;   /* move and draw solid line */

	char *drawBoxSequence;           /* draw box in current style */
	char *drawSolidBoxSequence;      /* draw a solid box */

	char *beginPlygnSequence;        /* begin polygon in cur. fill style */
	char *beginSolidPlygnSequence;   /* begin solid polygon */
	char *sendPlygnVertex;           /* define one point in polygon */
	char *endPlygnSequence;          /* terminate polygon sequence */

	char *drawCircleSequence;        /* draw a circle in solid line style */

	char *rotateTextSequence;        /* rotate graphic text */
	char *graphicsTextStart;         /* begin graphic text */
	char *graphicsTextEnd;           /* terminate graphic text */
	char *replaceON;                 /* turn on destructive text mode */
	char *overstrikeON;              /* turn on overstriking text mode */
	char *writePixel;                /* write one pixel in current color */

	char *setALUEOR;                 /* set ALU mode to EOR */
	char *setALUNOR;                 /* set ALU mode to NOR */
	char *setALUOR;                  /* set ALU mode to OR */
	char *setALUJAM;                 /* set ALU mode to JAM or REPLACE */

	char *blinkerON;                 /* make a color blink */
	char *blinkerOFF;                /* turn off a blinking layer */

	char *rastCopyStart;             /* begin raster copy sequence */
	char *rastCopyEnd;               /* terminate raster copy sequence */
	char *rastCopyDest;              /* define raster copy destionation */
	char *rastCopySource;            /* define raster copy source area */

	char *fillDefineStart;           /* begin defining a fill style */
	char *fillDefineFormat;          /* define bit array of row/column */
	char *fillDefineEnd;             /* terminate fill style definition */
	char *setFillPattern;            /* set current fill pattern */
	char *setSolidFillPattern;       /* set current fill pattern to solid */
	};


    struct mfb {               /* MFB DATA STRUCT */
```

```
                         /*
                          * INTEGERS FIELDS
                          */
/* used for decode */    int lastX,lastY;         /* for Tektronix encoding */
                         int X,Y,Z,T;             /* parameter list */

/* Resolution */         int maxX;                /* horizontal resolution */
                         int maxY;                /* vertical resolution */
                         int maxColors;           /* maximum number of colors */
                         int minOffScreenX;       /* left of off screen memory */
                         int minOffScreenY;       /* bottom of off screen mem. */
                         int offScreenDX;         /* length of off screen mem. */
                         int offScreenDY;         /* width of off screen mem. */

/* Video Layer Table */  int maxIntensity;        /* max RGB or LS intensity */
                         int lengthOfVLT;         /* number of bit planes */

/* Pointing Device */    int buttonMask[12];      /* returned button masks */
                         int numberOfButtons;     /* 12 maximum */

/* Keyboard Control */   int keyboardYOffset;
                         int keyboardXOffset;

/* Line Drawing */       int lineDefineLength;    /* number of bytes in array */
                         int maxLineStyles;       /* number of line styles */

/* Text font */          int fontHeight;          /* font height in pixels */
                         int fontWidth;           /* font width in pixels */
                         int fontXOffset;
                         int fontYOffset;

/* Blinkers */           int maxBlinkers;         /* number of blinkers */

/* Fill Patterns */      int fillDefineHeight;    /* number of byte rows */
                         int fillDefineWidth;     /* number of byte columns */
                         int maxFillPatterns;     /* number of fill patterns */

/* Current variables */  int cursorColor1Id;      /* blinked cursor color ID */
                         int cursorColor2Id;      /* unblinked cursor color ID */
                         int fgColorId;           /* cur. foreground color ID */
                         int fillPattern;         /* cur. fill pattern ID */
                         int lineStyle;           /* cur. line style ID */
                         int channelMask;         /* cur. write mask */
                         int readMask;            /* cur. read mask */
                         int numBlinkers;         /* cur. number of blinkers */
                         int textMode;            /* text mode (1=dest, 0=rep) */
                         int stipplePattern[8];   /* cur. stipple pattern */

                         int fileDesc;            /* desc. for graphics device */
                         int numTtyBuffer;        /* cur. chars in output queue */
```

```
                        /*
                         * POINTERS
                         */
/* for tty's only */    char *name;             /* device name */
                        char *terminalName;     /* name of MFBCAP entry */

/* I/O routines */      int (*dsply_getchar)();
                        int (*kybrd_getchar)();
                        int (*dsply_ungetchar)();
                        int (*kybrd_ungetchar)();
                        int (*outchar)();
                        int (*outstr)();


                        /*
                         * BOOLEANS
                         */
                        Bool initializedBool;

/* Video Layer Table */ Bool vltBool;
                        Bool vltUseHLSBool;

/* Channel Mask */      Bool channelMaskBool;
                        Bool readMaskBool;

/* Pointing Device */   Bool PointingDeviceBool;
                        Bool buttonsBool;
                        Bool readImmediateBool;

/* Keyboard Control */  Bool keyboardBool;

/* Line Drawing */      Bool linePatternDefineBool;
                        Bool reissueLineStyleBool;

/* Polygon Drawing */   Bool filledPlygnBool;

/* Text font */         Bool textPositionableBool;
                        Bool textRotateBool;
                        Bool replaceTextBool;
                        Bool overstrikeTextBool;

/* Blinkers */          Bool blinkersBool;

/* Raster Copy */       Bool rastCopyBool;
                        Bool rastRSCSFBool;

/* Fill Patterns */     Bool fillPtrnDefineBool;
                        Bool fillDefineRowMajorBool;

/* for tty's only */    Bool litout;
                        Bool raw;
                        Bool ttyIsSet;
```

```
        /*
         * STRUCTURES
         */
        struct mfbformatstrs strings;           /* format strings */

        WINDOW currentWindow;                   /* current window */
        VIEWPORT currentViewport;               /* current viewport */

#ifndef vms
        /* graphics device ttyb struct */
        MFBSAVETTYB graphTtyb;

        /* keyboard ttyb struct if graphics device does not have a kybrd */
        MFBSAVETTYB kybrdTtyb;

        /* tty status ints */
        MFBSAVESTAT oldstat;
#endif


        /*
         * CHARACTERS
         */
        char deviceType;                        /* TTY=tty, HCOPY=hard copy */
        char strBuf[BUFSIZE];                   /* storage for format strings */
        char ttyBuffer[TTYBUFSIZE];             /* tty output buffer */
        };

typedef struct mfb MFB;
```

## EXAMPLE

The following C program is a simple example that uses several *MFB* routines. The terminal type is assumed to be the first command line argument. This program will display several triangles in different line styles, display at four angles of rotation the text that is returned from *MFBKeyboard*, draw a solid line between two points, and draw an arc clipped to a rectangle.

```
    #include <cad/mfb.h>

    main(argc, argv)
        int argc;
        char *argv[];
        {
        int i, j, k, m;
        int X1, X2, Y1, Y2;
        int numcolors;
        int numlinestyles;
        int button;
        int error;
```

```
MFB *mfb, *MFBOpen();
MFBPATH *pp;
char key;
char *TypeIn;

/* open graphics device */
mfb = MFBOpen(argv[1], NULL, &error);

/* exit on any error */
if(error < 0) {
      fprintf(stderr, "error:  %s\n", MFBError(error));
      exit(0);
      }

/* get device information */
X1 = MFBInfo(MAXX)/2;
X2 = X1/2;
Y1 = MFBInfo(MAXY)/30;
numcolors = MFBInfo(MAXCOLORS);
numlinestyles = MFBInfo(MAXLINESTYLES);
if(numcolors > 7)
      numcolors = 7;
if(numlinestyles > 7)
      numlinestyles = 7;

/* draw pyramid of lines in different line styles */
for(j=0; j<28; ++j) {
      /* set color (increment by one to prevent 'invisible' lines) */
      k = j % (numcolors - 1);
      MFBSetColor(k + 1);

      /* define and set line style */
      m = j % numlinestyles;
      MFBDefineLineStyle(m, j * 6);
      MFBSetLineStyle(m);

      /* draw pyramid */
      MFBLine(0, 0, X2, Y1*j + 2*j);
      MFBLine(X1, 0, X2, Y1*j + 2*j);
      }

/* flush output */
MFBUpdate();
```

```
/* test of MFBKeyboard */
MFBText("Test of MFBKeyboard.", X2, Y1*28, 0);
TypeIn = MFBKeyboard(X1, 5, 0, 1);
MFBText(TypeIn, X1, 70, 0);
MFBText(TypeIn, X1, 70, 90);
MFBText(TypeIn, X1, 70, 180);
MFBText(TypeIn, X1, 70, 270);
MFBUpdate();
sleep(3);

/* test of MFBPoint */
MFBSetColor(1);
MFBFlood();
MFBSetColor(0);
MFBText("Test of MFBPoint.", X2, Y1*28, 0);
MFBPoint(&X1, &Y1, &key, &button);
MFBPoint(&X2, &Y2, &key, &button);
MFBSetLineStyle(0);
MFBLine(X1, Y1, X2, Y2);

/* draw outline of box to contain arc */
MFBLine(100, 100, 100, 350);
MFBLine(100, 100, 370, 100);
MFBLine(370, 100, 370, 350);
MFBLine(100, 350, 370, 350);

/* test of MFBArcPath and MFBClipArc */
i = 0;
pp = MFBClipArc(MFBArcPath(70, 70, 200, 0, 0, 30), 100, 100, 370, 350);
while(pp[i].nvertices != 0 && i < 4) {
      MFBDrawPath(&pp[i]);
      i++;
      }

/* flush output and wait */
MFBUpdate();
sleep(6);

MFBClose();
}
```

## DEBOUNCING THE POINTING DEVICE

The following C program is another example of using *MFB* routines that demonstrates several methods of debouncing the pointing device. Several graphics terminals can return bogus pointing reports that can be serious and annoying in some applications. Identifying these bogus reports is very terminal dependent (e.g. the AED 512 returns bad button masks, the Metheus 400 returns negative coordinates, etc.), and it is therefore necessary to use all possible tests.

```
#include <cad/mfb.h>
#ifdef vms
#include <timeb.h>
#else
#include <sys/timeb.h>
#endif

/*
 * This is the minimum time in milliseconds
 * between accepted pointing events.
 */
#define DEBOUNCETIME 100

/*
 * we keep track of the time between pointing
 * events to debounce the cursor
 */
static long LastPointTime = 0;

/*
 * routine to read and debounce pointing device.
 */
point(pointX,pointY,Key,Mask)
    int *pointX,*pointY,*Mask;
    char *Key;
    {
    struct timeb now;
    long newtime;
    int X,Y,Buttons;
    char KeyTyped;

    SetDebounceTime();
    /* Loop until DEBOUNCETIME has passed */
    while(True) {

        /* Loop until valid report is received */
        while(True) {

            /* Get pointing event */
            MFBPoint(&X,&Y,&KeyTyped,&Buttons);
            /* Was a character typed? */
            if(Key != 0)
                break;

            /* Does the pointing device have buttons? */
            if(MFBInfo(POINTINGBUTTONS)){
                /*
                 * Test button masks and vicinity of coordinate.
                 * Assume a four button mouse.
                 */
                if((Buttons == MFBInfo(BUTTON1) ||
                    Buttons == MFBInfo(BUTTON2) ||
                    Buttons == MFBInfo(BUTTON3) ||
                    Buttons == MFBInfo(BUTTON4)) &&
                    (X < MFBInfo(MAXX) && X > 0 &&
                    Y < MFBInfo(MAXY) && Y > 0))
                    break;
                }
            }
        ftime(&now);
        newtime = 1000 * now.time + now.millitm;
        if((newtime - LastPointTime) < DEBOUNCETIME) continue;
        SetDebounceTime();
        }
    *pointX = X;
    *pointY = Y;
    *Key = KeyTyped;
    *Mask = Buttons;
    }


SetDebounceTime(){
    struct timeb now;
    ftime(&now);
    LastPointTime = 1000 * now.time + now.millitm;
    }
```

## NOTES

On some systems, *MFB* is contained in /usr rather than ~cad.

*MFB* will also compile to run under *VMS* (a trademark of Digital Equipment Corp.) or any other operating system. However, special I/O routines such as those in ~cad/src/mfb/vmsio.c must be provided for *MFB* to function properly.

*MFB* was written to be utmost UNIX compatible and consistent with the style of the C programming language. For example, a control sequence always begins with a call to an *(MFB)Open* routine and is terminated by a call to a *(MFB)Close* routine. Another example is the provision of the *MFBHalt* routine that is intended primarily for the handling of the SIGTSTP signal. One possible exception to the style of C is the use of a global output descriptor that is set by a call to the *SetCurrentMFB* routine, as opposed to passing the output descriptor as an argument to ever active function call.

*MFB* was initially aimed toward the modeling of lower performance graphics terminals (e.g., there is currently no support of segments or definable windows and viewports at the device level). As a result, programs that use *MFB* are likely to work on the low performance (least expensive) graphics terminals as well as on the more expensive devices.

## BUGS

Raster (hard copy) output is not yet implemented.

## FUTURE ENHANCEMENTS

Future modifications to *MFB* may include the following:

Extension to hard copy graphics devices.

Definable vector and raster character fonts.

Improved cursor support including a definable cursor font, cursor tracking by the host, cursor-on/cursor-off capability, and cursor report without event.

Window/viewport geometry clipping by the terminal if the device possesses that capability.

Bit block transfer (BitBlt).

## FILES

~cad/lib/mfbcap
~cad/include/mfb.h
~cad/lib/mfb.a
~cad/src/mfb

## SEE ALSO

mfbcap(5), termcap(5), curses(3), more(1), kic(CAD1)

## AUTHOR

Giles Billingsley
Ken Keller

**STATUS**

The following is a list of the terminals that will currently work with *MFB:*

| | |
|---|---|
| 4014 | Tektronix 4014 with thumbwheels |
| 4113 | Tektronix 4112/4113 with thumbwheels or tablet |
| AED5 | AED 512 with joystick or tablet |
| AED7 | AED 767 with joystick or tablet |
| 2648 | HP 2648 black and white grahpics terminal with tablet |
| 9872 | HP 9872 color pen plotter |
| D125 | DEC VT125 black and white graphics terminal |

The following table lists routines that depend on device capabilities and may not work on all graphics devices. Other *MFB* routines that are not listed below will work for all devices.

| Routine | 4014 | 4113 | AED5 | AED7 | 2648 | 9872 | D125 |
|---|---|---|---|---|---|---|---|
| MFBSetLineStyle | X | X | X | X | X | X | X |
| MFBSetFillPattern | | X | X | X | X | X | X |
| MFBSetChannelMask | | | X | X | | | |
| MFBSetReadMask | | | X | X | | | |
| MFBSetColor | | X | X | X | X | X | X |
| MFBSetTextMode | | X | | | | | |
| MFBSetALUMode | | X | | | X | | |
| MFBSetCursorColor | | | X | X | | | |
| MFBSetRubberBanding | | X | | | X | | |
| MFBSetBlinker | | | X | X | | | |
| MFBDefineColor | | X | X | X | | | |
| MFBDefineFillPattern | | X | X | X | X | | |
| MFBDefineLineStyle | | | X | X | X | | |
| MFBMoveTo | X | X | X | X | X | X | X |
| MFBDrawLineTo | X | X | X | X | X | X | X |
| MFBLine | X | X | X | X | X | X | X |
| MFBDrawPath | X | X | X | X | X | X | X |
| MFBBox | X | X | X | X | X | X | X |
| MFBPolygon | X | X | X | X | X | X | X |
| MFBFlood | | X | X | X | X | | X |
| MFBPixel | X | X | X | X | X | X | X |
| MFBCircle | X | X | X | X | X | X | X |
| MFBFlash | X | X | X | X | X | X | X |
| MFBArc | X | X | X | X | X | X | X |
| MFBText | X | X | X | X | X | X | X |
| MFBPoint | X | X | X | X | X | | |
| MFBKeyboard | X | X | X | X | X | | X |
| MFBMore | | X | X | X | X | | X |
| MFBScroll | | X | X | X | X | | X |

# Appendix E

# The MFBCAP Programmer's Manual

The Section 5 UNIX manual pages for the MFBCAP graphics terminal database file are contained in this appendix.

### NAME

mfbcap – graphics terminal capability data base

### SYNOPSIS

~cad/lib/mfbcap

### DESCRIPTION

*MFBCAP* is a data base describing graphics terminals, used, *e.g.*, by *kic*(1) and *mfb*(3). Terminals are described in *MFBCAP* by defining a set of capabilities that they have, and by describing how operations are performed. Output delays and initialization sequences are also included in *MFBCAP*.

Entries in *MFBCAP* consist of a set of comma (,) separated fields. Entries may continue onto multiple lines by beginning a continuation line with either a tab or space character. The first entry for each terminal gives the names by which the terminal is known, separated by vertical bar (|) characters. The first name is always 2 characters long, the second name given is the most common abbreviation for the terminal, and the last name given should be a long name fully identifying the terminal. The second name should contain no blanks; the last name may contain blanks for readability. For compatability with other operating systems, it is recommended that the device names use all UPPER CASE LETTERS.

### CAPABILITIES

The Parms column indicates which of the four possible parameters are used in the encoding/decoding of string variable. Upper case letters signify that the parameter value is passed to the *mfb(3)* routine, and lower case letters identify values that are returned.

| **Name** | **Type** | **Parms** | **Description** |
|---|---|---|---|
| 8BB | boolean | | transmit in 8 Bit Binary using LITOUT |
| ALUEOR | string | | set ALU writing mode to Exclusive OR |
| ALUJAM | string | | set ALU writing mode to JAM (replace mode) |
| ALUNOR | string | | set ALU writing mode to NOR |
| ALUOR | string | | set ALU writing mode to OR |
| APT | boolean | | Accurately Positionable Text |
| BELL | string | | ring the terminals BELL |
| BLD | boolean | | BLinkers Definable |
| BLE | string | | BLinkers End |
| BLS | string | XYZT | BLinkers Start<br>X = off color ID<br>Y = red/hue intensity when blinked<br>Z = green/lightness intensity when blinked<br>T = blue/saturation intensity when blinked |
| BU1 | numeric | | value returned by BUtton 1 of pointing device |
| BU2 | numeric | | value returned by BUtton 2 of pointing device |
| BU3 | numeric | | value returned by BUtton 3 of pointing device |
| BU4 | numeric | | value returned by BUtton 4 of pointing device |
| BU5 | numeric | | value returned by BUtton 5 of pointing device |
| BU6 | numeric | | value returned by BUtton 6 of pointing device |
| BU7 | numeric | | value returned by BUtton 7 of pointing device |
| BU8 | numeric | | value returned by BUtton 8 of pointing device |
| BU9 | numeric | | value returned by BUtton 9 of pointing device |
| BU10 | numeric | | value returned by BUtton 10 of pointing device |
| BU11 | numeric | | value returned by BUtton 11 of pointing device |
| BU12 | numeric | | value returned by BUtton 12 of pointing device |

| | | | |
|---|---|---|---|
| DBS | string | XYZT | Draw Box Sequence<br>X = lower left<br>Y = lower bottom<br>Z = upper right<br>T = upper top |
| DCS | string | XYZ | Draw Circle Sequence<br>X = center x coordinate<br>Y = center y coordinate<br>Z = radius of circle |
| DFP | boolean | | Definable Fill Patterns |
| DLP | boolean | | Definable Line Patterns |
| DLS | string | XYZT | Draw Line Sequence<br>X,Y = start coordinate<br>Z,T = end coordinate |
| DLT | string | XY | Draw Line To (x,y) sequence<br>X,Y = next current graphics position |
| DSL | string | XYZT | Draw Solid Line sequence<br>X,Y = start coordinate<br>Z,T = end coordinate |
| DSLT | string | XY | Draw Solid Line To (x,y) sequence<br>X,Y = next current graphics position |
| DSB | string | XYZT | Draw Solid Box sequence<br>X = lower left<br>Y = lower bottom<br>Z = upper right<br>T = upper top |
| FDE | string | X | Fill pattern Define End<br>X = style ID |
| FDF | string | XY | Fill pattern Define Format<br>X = style ID<br>Y = one 8 bit row/col of the fill pattern array |
| FDH | numeric | | Fill pattern Define Height in rows |
| FDR | boolean | | Fill pattern Define Row major |
| FDS | string | X | Fill pattern Define Start<br>X = style ID |
| FDW | numeric | | Fill pattern Define Width in columns |
| FPOLY | boolean | | terminal is capable of Filled POLYgons |
| GCH | numeric | | Graphics Character Height |
| GCS | string | | Graphics Clear Screen (in current color) |
| GCW | numeric | | Graphics Character Width |
| GFS | string | | Graphics Finish String |
| GIS | string | | Graphics Initialization String |
| GTE | string | | Graphics Text End |
| GTH | numeric | | Graphics Text Height offset |
| GTO | boolean | | Graphics Text Overstrikes old text |
| GTR | boolean | | Graphics Text Replaces old text |
| GTS | string | XYZ | Graphics Text Start<br>X,Y = lower left coordinate of text string<br>Z = number of characters in text string |
| GTW | numeric | | Graphics Text Width offset |
| HLS | boolean | | convert RGB color definitions to HLS |
| ICS | string | | Initialize predefined Color Styles |

| | | | |
|---|---|---|---|
| IFP | string | | Initialize predefined Fill Patterns |
| ILS | string | | Initialize predefined Line Styles |
| KYB | string | | KeYboard Backspace sequence |
| KYBRD | boolean | | Terminal has a KeYBoaRD |
| KYE | string | | KeYboard End sequence |
| KYS | string | XY | KeYboard Start sequence<br>X,Y = lower left coordinate of keyboard window |
| KYX | numeric | | KeYboard X offset |
| KYY | numeric | | KeYboard Y offset |
| LDE | string | X | Line Define End<br>X = style ID |
| LDF | string | XY | Line Define Format<br>X = style ID<br>Y = 8 bit fill pattern |
| LDL | numeric | | Line Define Length (in bytes) |
| LDS | string | X | Line Define Start<br>X = style ID |
| MCE | string | | device behaves like the following MfbCap Entry |
| MCL | numeric | | Maximum number of Colors |
| MFP | numeric | | Maximum number of Fill Patterns |
| MLS | numeric | | Maximum number of Line Styles |
| MPS | string | XY | Move Pen Sequence<br>X,Y = coordinate to move graphics cursor |
| MXC | numeric | | Maximum X Coordinate |
| MYC | numeric | | Maximum Y Coordinate |
| NBL | numeric | | Number of BLinkers |
| NPB | numeric | | Number of Pointing device Buttons |
| OFFDX | numeric | | length of OFF screen memory in X Direction |
| OFFDY | numeric | | length of OFF screen memory in Y Direction |
| OFFMX | numeric | | minimum X coodinate of OFF screen Memory |
| OFFMY | numeric | | minimum Y coodinate of OFF screen Memory |
| OMO | string | | Overstrike text Mode On sequence |
| PDB | boolean | | Pointing Device has Buttons |
| PDE | string | | Pointing Device End |
| PDF | string | xyzt | Pointing Device coordinate Format<br>x,y = input coordinate<br>z = key pushed<br>t = button mask |
| PDR | string | | Pointing Device initiate Read |
| PDS | string | | Pointing Device Start |
| PLE | string | XY | PoLygon End sequence<br>X,Y = first coordinate in the polygon sequence |
| PLS | string | XYZ | PoLygon Start sequence<br>X,Y = first of Z coordinates<br>Z = number of coordinates |
| PLSOL | string | XYZ | PoLygon start sequence for SOLid fill<br>X,Y = first of Z coordinates<br>Z = number of coordinates |

| | | | |
|---|---|---|---|
| PLV | string | XY | send PoLygon Vertex sequence<br>X,Y = next coordinate in the polygon sequence |
| POD | boolean | | terminal has POinting Device |
| PRBOFF | string | | disable Pointing device Rubber Banding |
| PRBON | string | | enable Pointing device Rubber Banding |
| PRI | boolean | | Pointing Read Immediately returns coordinates |
| RAW | boolean | | drive device in RAW mode |
| RLS | boolean | | Reissue Line Style before each line |
| RMO | string | | Replace text Mode On sequence |
| ROT | boolean | | ROTatable graphics text |
| RTS | string | X | Rotate Text Sequence<br>X = angle of rotation in degrees (-360 <= X <= 360) |
| RSCPE | string | | RaSter CoPy End sequence |
| RSCPS | string | | RaSter CoPy Start sequence |
| RSCSF | boolean | | transmit RaSter Copy Source coordinate First |
| RSDST | string | XYZT | RaSter copy DeSTination sequence<br>X,Y = destination coordinate<br>Z,T = length,width of area to be copied |
| RSSRC | string | XYZT | RaSter copy SouRCe sequence<br>X,Y = source coordinate<br>Z,T = length,width of area to be copied |
| SCS | string | X | Set Color Style<br>X = new color ID |
| SFP | string | X | Set Fill Pattern<br>X = new fill pattern ID |
| SLS | string | X | Set Line Style<br>X = new line style ID |
| SRM | string | X | Set video Read Mask<br>X = channel read mask |
| SSFP | string | X | Set Solid Fill Pattern<br>X = new fill pattern ID |
| SSLS | string | X | Set Solid Line Style<br>X = new line style ID |
| TTY | boolean | | device is a TTY |
| VLT | boolean | | Video Lookup Table present |
| VTE | string | XYZT | Video Table Entry<br>X = color ID of new entry<br>Y = red/hue intensity<br>Z = green/lightness intensity<br>T = blue/saturation intensity |
| VTI | numeric | | Video Table maximum Intensity |
| VTL | numeric | | VLT Length expressed as number of bit planes |
| VWM | string | X | Video Write Mask<br>X = channel write mask |
| WPX | string | XY | Write PiXel at coordinate XY |

### A Sample Entry

The following entry describes the HP 2648. (This particular 2648 entry may be outdated, and is used as an example only.)

```
#
# HP2648 with keyboard cursor control
#
h0|H0|2648|HP2648|HP2648A|Hewlett-Packard 2648A,
          TTY, APT, MXC#719, MYC#359, MCL#2, MFP#8, MLS#2,
          GTO, DFP, DLP,
          MPS=E*pa%X%d,%Y%dZ,
          DLT=E*pf%X%d,%Y%dZ, RLS,
          DBS=E*m3b%X%d,%Y%d,%Z%d,%T%dE,
          DSL=E*m1BE*pa%X%d,%Y%d,%Z%d,%T%dZ,
          DLS=E*pa%X%d,%Y%d,%Z%d,%T%dZ,
          WPX=E*pa%X%d,%Y%d,%X%d,%Y%dZ,
          PLS=E*pa%X%d,%Y%d,
          PLV=,%X%d,%Y%d,
          PLE=,%X%d,%Y%dZ,
          LDL#1, LDF=E*m%Y%d 1C,
          GCS=E*d%X%+A%c$<#500>, GCH#11, GCW#7,
          GFS=EHEJE*mRE*dlaeD$<#2500>,
          GIS=E*mRE*dlafC$<#3500>,
          GTE=E*dT, GTH#1, GTW#1,
          GTS=E*pa%X%d,%Y%dZE*dS,
          KYBRD, KYB=^H,
          KYS=E*pa%X%d,%Y%dZE*m4aE*dS,
          KYE=E*dT, KYX#1, KYY#1,
          SFP=E&f%X%+1%cE21,
          SCS=E*m%X%+1%cA,
          SLS=E*m2B,
          FDH#8, FDW#8, FDR, FDF= %Y%3,
          FDS=E&f1a%X%+1%ck36LE*m,
          FDE=D$<#90>,
          POD, PDR=E*s4  21, PDS=E*dK,
          PRBON=E*dM, PRBOFF=E*dN,
          PDF=+%d,%X+%d,%Y%3%Z%c, PDE=  21E*dL,
```

Capabilities in *MFBCAP* are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal, and string capabilities which give a sequence that can be used to perform particular terminal operations.

### Types of Capabilities

All capabilities have an identifying code. For instance, because the HP2648 has "accurately positionable text" ( *i.e.*, graphics text may be positioned with lower left corner at any pixel on the screen ) is indicated by the boolean **APT**. Hence the description of the HP2648 includes **APT**. Numeric capabilities are followed by the character '#' and then the value. Thus **MXC** which specifies the maximum value of the X coordinate on the terminal viewport gives the value '719' for the HP2648.

**Formatting String Capabilities**

String variables have a formatting capability to be used for encoding numbers into ASCII strings and decoding ASCII strings into numbers. An example of the former is the capability **DBS** ( for **D**raw **B**ox **S**equence ), which takes four numbers (X, Y, Z, and T) and generates the proper sequence to draw a box from the lower left corner (X,Y) to the upper right corner (Z,T). An example of a string decode is the capability **PDF** ( for **P**ointing **D**evice **F**ormat ), which takes an ASCII string from the input stream and extracts from it an x and y coordinate, a key (if one was pushed) and a buttonmask (if a cursor button was pushed).

**String Formatting**

The string variables have a formatting capability which uses four variables (X, Y, Z, and T) to generate a formated string (with *MFBGenCode),* or generates four variables (X, Y, Z, and T) from a formated string (with *MFBDecode).* Two temporary registers represented by the letters **R** and **r** are available. All operations begin with a percent sign '%', and they are listed below:

| **Com** | **Command Description encode/(decode)** |
|---|---|
| %X | set value/(X variable) to the X variable/(value). |
| %Y | set value/(Y variable) to the Y variable/(value). |
| %Z | set value/(Z variable) to the Z variable/(value). |
| %T | set value/(T variable) to the T variable/(value). |
| %C | set value to the current foreground color ID. |
| %F | set value to the current fill pattern ID. |
| %L | set value to the current line style ID. |
| %d | output/(input) value in variable length decimal format |
| %2 | output/(input) value converting to/(from) two decimal digits. |
| %3 | output/(input) value converting to/(from) three decimal digits. |
| %c | output/(input) least significant byte of value withoutconversions. |
| %h1 | output/(input) least significant four bits converting to/(from) one ASCII hex character. |
| %h2 | output/(input) least significant byte converting to/(from) two ASCII hex characters. |
| %h3 | output/(input) least significant twelve bits converting to/(from) three ASCII hex characters. |
| %h4 | output/(input) least significant sixteen bits converting to/(from) four ASCII hex characters. |
| %o1 | output/(input) least significant three bits converting to/(from) one ASCII octal character. |
| %o2 | output/(input) least significant six bits converting to/(from) two ASCII octal characters. |
| %o3 | output/(input) least significant nine bits converting to/(from) three ASCII octal characters. |
| %o4 | output/(input) least significant twelve bits converting to/(from) four ASCII octal characters. |
| %o5 | output/(input) least significant fifteen bits converting to/(from) five ASCII octal characters. |
| %o6 | output/(input) least significant sixteen bits converting to/(from) six ASCII octal characters. |
| %t1 | output/(input) X and Y in Tektronix format. |

%t2 output/(input) Z and T in Tektronix format.

%t3 output X and R in Tektronix format (MFBGenCode only).

%t4 output R and Y in Tektronix format (MFBGenCode only).

%t5 output R and r in Tektronix format (MFBGenCode only).

%ti output/(input) value in Tektronix integer format.

%tr output value in Tektronix real format.

%R store/(retrieve) value in temporary register 1.

%r store/(retrieve) value in temporary register 2.

%+x add x to value.

%-x subtract x from value.

%*x multiply value by x.

%/x divide value by x.

%>>x shift value right by x bits.

%<<x shift value left by x bits.

%|x OR x with value.

%&x AND x with value.

%^x EOR x with value.

%=x set value equal to x.

%ax set value equal to the absolute value of x.

%~ Complement value ( 1's complement ).

%@ output a single null character (MFBGenCode only).

%% gives '%'.

%B BCD (2 decimal digits encoded in one byte).

%D Delta Data (backwards bcd).

Where x can be:

(1) One byte - the numeric value of this byte is used as x.

(2) The character "#" followed by a decimal integer value for x.

(3) The character "%" followed by C, F, L, X, Y, Z, T, r, or R - the value of C, F, L, X, Y, Z, T, r or R is used.

The command formats are similar to those found in *termcap(5)* or *terminfo(5)*, but are more complicated due to the more rigorous requirements of graphics terminals.

**Preparing Descriptions**

We now outline how to prepare *MFBCAP* descriptions of graphics terminals. The most effective way to prepare a terminal description is to build up a description gradually, using partial descriptions with simple *mfb(3)* test routines to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the *MFBCAP* file to describe it. To easily test a new terminal description you can set the environment variable **MFBCAP** to a pathname of a file containing the description you are working on. After setting the environment variable, any program that uses *mfb(3)*, *e.g.*, *kic*, will look at the pathname defined by the environment variable instead of *~cad/lib/mfbcap*.

### Delays

Delays may be embedded anywhere in a string capability and is distinguished by the **$<** and **>** brackets. The number contained within these brackets describes the delay in milliseconds to be generated and must conform to the above description for the variable 'x' ( *e.g.*, an integer constant must be preceded by the character "#"). Before each delay, the output buffer is flushed.

### Basic Capabilities

The number of pixels on a horizontal row of the display is given by the **MXC** numeric capability, and the number of pixels in a vertical column is given by the **MXY** capability. The number of colors available on the display is specified by the **MCL** capability. For black and white terminals, such as the HP2648, the **MCL** capability is defined as two. The maximum number of stipple fill patterns and line styles is given by the **MFP** and **MLS** numeric capabilities respectively.

Off screen memory refers to an area of the viewport in pixel coordinates which is not displayed. The lower, left corner of the off screen memory is specified by the **OFFMX** and **OFFMY** numeric capabilities. The horizontal length of the off screen memory is specified by the **OFFDX** numeric capability, and the vertical width of the off screen memory is specified by the **OFFDY** numeric capability.

*MFBCAP* allows two sequences for initializing and uninitializing the terminal. The first initialization string sent to the terminal is given by the **GIS** format string. This will be the first sequence sent to the graphics device. The graphics finish/termination string is given by the **GFS** format string. This will be the last sequence sent to the graphics device.

The initialization character sequences for color styles, fill styles, and line styles are defined respectively by the **ICS**, **IFP**, and **ILS** format strings.

The sequence to ring the terminals bell or alarm is defined by the **BELL** string and defaults to control-G.

### Setting Colors and Styles

The character sequence for setting the current foreground color is defined by the **SCS** format string. All subsequent geometries will be drawn in this color. The format for setting the current line style is given by the **SLS** format string, and the format for setting the current fill style is given by the **SFP** format string. All subsequent lines, boxes, and polygons will be drawn with these styles. *MFBCAP* assumes that style zero defines a solid line and fill pattern. If this is not the case for a particular frame buffer, or the format for setting a solid line or fill style is inconsistent with that for other line styles, such as is the case for the HP9872, a character sequence for setting the solid line or fill style is defined by the **SSLS** and **SSFP** format strings respectively. If it is necessary for the current line style to be reissued before a line is drawn (as is the case for the HP 2648), then the **RLS** boolean must be present in the *MFBCAP* entry.

### Basic Geometries

The character sequence for moving the current graphics position to a x,y pixel coordinate is defined by the **MPS** format string. The format for drawing a line in the current line style from the current graphics position to a x,y pixel coordinate is defined by the **DLT** format string. The character sequence to draw a line in the current line style from a x,y pixel coordinate to a z,t pixel coordinate is defined by the **DLS** format string. If the command for drawing a solid line is different from that for a non-solid line, the character sequence to draw a solid line from a x,y pixel coordinate to a z,t pixel coordinate may be defined by the **DSL** string capability. The format for drawing a solid line from the current graphics position to a x,y pixel coordinate is defined by the **DSLT** format string.

The sequence for drawing a box in the current foreground color from the lower left x,y pixel coordinate to the upper right z,t pixel coordinate is specified by the **DBS** format string. Because some terminals, such as the Tektronix 4113, have special raster commands for drawing solid boxes, a format for drawing solid boxes may be specified by the **DSB** format string.

The format for setting a pixel in the current color at the x,y pixel coordinate is defined by the **WPX** format string.

The format for drawing a circle with its center at the x,y pixel coordinate and having a radius of z pixels is defined by the **DCS** format string.

The format for clearing the entire screen to the current color is given by the **GCS** format string. If there is no such command sequence, it may be substituted by the command sequence that will write a solid box in the current color over the entire screen.

There are three format strings for defining the terminal's polygon command sequence. First the **PLS** starting sequence is used to define x,y as the first of z pixel coordinates. This character sequence will be followed by z-1 occurrences of the **PLV** format string which defines the remaining vertices of the polygonal path. Finally, an ending sequence that is defined by the **PLE** format string terminates the polygon sequence. For terminals which have inconsistent formats for drawing solid polygons, the **PLSOL** sequence may be used in place of the **PLS** sequence. If the terminal is capable of drawing a filled polygon in the current fill pattern, then the **FPOLY** boolean should appear in the *MFBCAP* entry.

**Video Layer Table**

If the terminal has a video layer table, then the **VLT** boolean must be present in the *MFBCAP* entry. *MFBCAP* assumes that the VLT uses the red-green-blue system for defining colors. If the **HLS** boolean capability is specified, then the RGB arguments become HLS (hue-lightness-saturation) values. The maximum intensity of red, green, or blue in the VLT (or the lightness or saturation if using the HLS system) is given by the **VTI** numeric capability. The format for setting a particular entry of the VLT is given by the **VTE** format string. The **VTL** numeric value can be used to define the length of the VLT in terms of the number of bit planes.

**Defining Styles**

The **LDS** string capability defines the sequence for (re)defining a line style corresponding to a particular style ID. The **LDF** format string is used to define an eight bit mask that represents the new line style. The **LDE** format string terminates the definition of the new line style.

The definition of a new fill pattern is more complicated than is the case for line styles. It is necessary to transform an eight by eight intensity array into whatever command syntax is required by the terminal. The **FDS** string capability is used to begin the (re)definition of a fill pattern. The **FDF** format string defines one row or column of the fill pattern using an eight bit mask (one row of the eight by eight intensity array). If the **FDR** boolean is present, then it is assumed that the fill pattern is being defined by rows in which case the **FDF** sequence is sent by the number of times defined by the **FDH** numeric capability. Otherwise, it is assumed that the fill pattern is defined by columns, and the **FDF** sequence is sent by the number of times defined by the **FDW** numeric capability. If, for example, the number of rows in the fill pattern is ten, the **FDF** sequence is first transmitted using each of the eight rows of the initial eight by eight intensity array, and then the sequence is sent twice using the first and second rows of the initial intensity array. The **FDE** format string terminates the definition of the new fill pattern.

**Raster Capabilities**

There are four format strings for defining the terminal's raster copy command sequence. First the **RSCPS** starting sequence is used to begin the raster copy command. This character sequence will be followed the **RSSRC** format string which defines the lower, left coordinate and length and width of the source area and the **RSDST** format string which defines the lower left coordinate and the length and width of the destination area. The **RSSRC** sequence appears first only if the **RSCSF** boolean is defined. Finally, an ending sequence that is defined by the **RSCPE** format string terminates the raster copy sequence.

**Graphic Text**

*MFBCAP* supports a single font graphic text. The height and width of the text font are given respectively by the **GCH** and **GCW** numeric capabilities. Graphics text is displayed with three format strings. A text string with z characters with a lower left justification at the x,y pixel coordinate is begun with the format string defined by **GTS**. This will be followed by the transmission of the z characters and terminated by the format string defined by **GTE**. The graphic text can offset from the current graphics position by setting the numeric capabilities **GTH** and **GTW**. The following figure demonstrates the assumed character font for the two characters "gh". The character "0" marks a pixel in the character font, and the character "X" marks the x,y pixel coordinate to which the two characters where justified. Note that the **GCH**, **GCW**, **GTH**, and **GTW** numeric capabilities must always be non-negative integers.

```
              _   . . . . . .     0 . . . . .
              |   .         .     0         .
              |   .         .     0         .
              |   . 0 0 0 0 .     0 0 0 0   .
              |   0         0     0       0 .
              |   0         0     0         0
      _     GCH   0         0     0         0
      |       |   X 0 0 0 0 0     0         0
      |       |   .         0     .         .
     GTH      |   .         0     .         .
      |       |   0         0     .         .
      |       |   . 0 0 0 0 .     . . . . . .
      -       -
                  | -GCW-   | -GTW- |
```

If the terminal supports rotatable graphic text, then the **ROT** boolean is present in the *MFBCAP* entry. If rotated text is desired the **RTS** character sequence is issued prior to the the **GTS** sequence and defines a rotation of x degrees, where x is between -360 and 360.

*MFBCAP* supports two graphic text modes. If the graphic text can be destructive, then the **GTR** boolean is present in the *MFBCAP* entry, and the **RMO** format string specifies the character sequence for entering the destructive graphic text mode. If the terminal has graphic text that can overstrike, then the **GTO** boolean is present in the *MFBCAP* entry, and the **OMO** format string defines the character sequence for entering the overstriking graphic text mode.

**Keyboard Control**

For terminals with special keyboard/cursor operations, *MFBCAP* provides a set of string capabilities for controlling keyboard input. The keyboard is initialized, and the current graphics position is moved to the x,y pixel coordinate by the **KYS** format string. The current graphics position can be offset upward from the above x,y pixel coordinate by setting values to the **KYX** and **KYY** numeric capabilities. The keyboard backspace sequence is defined by the **KYB** format string (the is NO default for the backspace format string). The keyboard is uninitialized by the **KYE** format string.

If the terminal does not have the above capabilities, a keyboard input routine, such as that used in *mfb(3)*, can use the terminals graphic text capabilities to echo keyboard characters on a command line.

**Pointing Device.**

If the terminal has a pointing device, then the **POD** boolean is present in the *MFBCAP* entry. If the pointing device has buttons, then the **PDB** boolean is set, the number of buttons is given by the **NPB** numeric capability, and the values returned by the respective buttons of the pointing device are defined by the **BU1** through **BU12** numeric capabilities.

The graphics pointing device is initialized with the **PDS** format string. The **PDR** format string places the terminal in a waiting mode until the first graphic input. When this event occurs, the locator event is decoded by the **PDF** format string. The graphic pointing device is uninitialized by the **PDE** format string.

If the **PRI** boolean is set, one character is read immediately after the pointing device initialization sequence **PDS** and before the pointing device is enabled by **PDR**. This is useful for terminals that have a cursor and can read its current position but do not have the capability of a graphic event ( *i.e.*, an x,y pixel coordinate that is read immediately after a key or button is pushed on the terminal).

The pointing device encoding format string **PDF** must assume that the pointing device will send one signature character. After the pointing is activated by the **PDR** format string, the first character transmitted from the terminal must be identical to the first character of the **PDF** format string. If the characters do not match, then the *MFBDecode* routine used by *mfb(3)* will return the first character that was transmitted by the terminal.

The **PRBON** string capability defines the character sequence to enable rubber banding of the pointing device, and **PRBOFF** disables the rubber banding. *MFBCAP* assumes that the center of rubber banding is the current graphics position that can be defined by the **MPS** format string defined above.

**Special Modes**

If the graphics device is to be handled as a TTY, then the **TTY** boolean must be present in the *MFBCAP* entry. If the graphics encoding can produce 8 bit, non-ASCII characters, then the **8BB** boolean must be included. If the graphics device is a TTY and is to be driven in a *RAW* mode, then the **RAW** boolean must be included. See the manual for *tty(4)*. Typically, this mode is used only if the device has no keyboard.

*MFBCAP* supports four ALU writing modes. These are the modes in which a pixel is updated when written over. The four possible modes are JAM (replace mode), OR, EOR, and NOR. The sequences for setting these modes are **ALUJAM**, **ALUOR**, **ALUEOR**, and **ALUNOR** respectively.

### Similar Terminals

If there are two very similar terminals, one can be defined as being like the other but with certain exceptions. The string capability **MCE** is given with the name of the similar terminal. The **MCE** must be the last capability defined in the entry, and the combined length of the two entries must not exceed 4096 characters. Because *mfb* routines scan the entry from left to right, and because the **MCE** entry is replaced by the corresponding entry, the capabilities given on the left override identical capabilities defined for the similar terminal. This is useful for defining different modes for a terminal, or for defining terminals with different peripherals.

## FILES

~cad/lib/mfbcap file containing terminal descriptions

## SEE ALSO

termcap(5), mfb(3), kic(CAD1)

## AUTHOR

Giles Billingsley

## BUGS

The total length of a single entry (excluding only escaped new lines) may not exceed 8192.

There is a restriction that allows a simple parser to be used for the *MFBCAP* file. The delimiter is assumed to be a comma that is not immediately preceded by a slash (\) character. String capabilities that terminate with a slash character (as is the case for the vt125) must therefore separate the delimiting comma and the slash character with a padding character.

Not all programs support all entries. There are entries that may not be used by any program.

# Appendix F

# KIC and Related Manual Pages

The Section 1 UNIX manual pages for KIC and all related programs are contained in this appendix.

## NAME

ciftokic – Translate a CIF file into KIC format.

## SYNOPSIS

**ciftokic [-L micperl] [-I ciffile] [-ahikns]**

## DESCRIPTION

*Ciftokic* translates a CIF file into the hierarchical directory format used by *KIC(CAD1)*. The -L option is used to specify the number of microns per lambda. The -I option is used to specify the name of the input CIF file. The remaining options are used to indicate the style of the CIF file:

**–k** The CIF file was created by *KICToCIF(1)* or *STREAMToCIF(1)* with standard Berkeley extensions and may include property lists using user extention 5.

**–h** The CIF file was created by IGS. An IGS symbol name is a CIF user extension which follows a DS command as in

9 PadIn;

**–i** The CIF file was created by Icarus. An Icarus symbol name is a CIF comment which follows a DS command as in

(9 PadIn);

**–q** The CIF file was created by Squid. A Squid symbol name is a CIF user extension which follows a DS command and specifies the full path name of the file as in

9 /usr/joe/layout/PadIn;

**–m** The CIF file is compatable with the mextra program which uses the symbol rename convention of Berkeley CIF and has the respective layer name in the label user extension as in *94 Text 1500 -9800 NM;*.

**–s** The CIF file was created by Sif. A Sif symbol name is a CIF comment which follows a DS command as in

(Name: PadIn);

**–a** The CIF file was created by Stanford. A Stanford symbol name is a CIF comment which follows a DS command as in

(PadIn);

**–b** The CIF file was created by NCA. A NCA symbol name is a CIF comment which follows a DS command as in

(PadIn);

**–n** The CIF file was created by none of the above.

*Ciftokic* will become an interactive program if the command line arguments are insufficient. Input and output do NOT default to standard I/O. If no command line arguments are given, *ciftokic* will first prompt you to identify the style of the CIF file as one of the options mentioned above. The program will next prompt you for the number of microns per lambda. Finally, *ciftokic* will ask the name of the input CIF file.

If the CIF file was generated by KIC, IGS, Icarus, Stanford, or Sif, each CIF symbol will become a KIC cell adopting the name of that CIF symbol. All CIF symbol numbers must be between 0 and 999 inclusive. The root cell is the very top of the hierarchy and will be the KIC cell named *Root*.

If the CIF file was not generated by any of the above programs, each CIF symbol will become a KIC cell with the name *SYMBOLxxx*, where *xxx* is the number of the respective CIF symbol.

**SEE ALSO**

kic(CAD1), kictocif(CAD1), KIC tutorial stored in ~cad/doc/kic.me

**AUTHOR**

Giles Billingsley

**BUGS**

The CIF file may not contain a symbol named *Root*. If the CIF file was created by *kictocif(1)* with a root hierarchy file named *Root*, then the CIF file will also contain a symbol named *Root* and will have to be edited for successful conversion.

*Ciftokic* does not check for duplicate symbol names in the CIF file.

## NAME

ciftostrm – Create a GDS II STREAM file from a CIF file.

## SYNOPSIS

**ciftostrm [-X lfile] [-Z libname] [-O streamfile] [-I ciffile] [-ahikns]**

## DESCRIPTION

*Ciftostrm* translates a CIF file into a GDS II STREAM file. The output STREAM file will always contain 100 database units per micron.

The **-O** and **-I** options are used to specify the output and input files, respectively. If the **-O** option is not employed, the name of the output STREAM file will be the name of the input *ciffile* appended with *'.str'.*

The **-X** option is used to specify the file name *lfile* of the CIF-STREAM layer table. The structure of the CIF-STREAM layer table is illustrated below:

```
(number of CIF layer names)
(CIF layer name 1)  (STREAM layer value)  (STREAM datatype value)
(CIF layer name 2)  (STREAM layer value)  (STREAM datatype value)
(CIF layer name 3)  (STREAM layer value)  (STREAM datatype value)
          .
          .
          .
         etc.
```

The STREAM layer and datatype values must be non-negative integers, and the CIF layer names should be unique. Comments may be appended to the CIF-STREAM layer table after the last CIF layer definition.

The name of the STREAM library file may be specified to be *lname* with the **-Z** option. The default library name is *CIFTOSTREAM* .

The remaining options are used to indicate the style of the CIF file:

**–k** The CIF file was created by *KICToCIF(1)* or *STREAMToCIF(1)* with the standard Berkeley extensions.

**–h** The CIF file was created by IGS. An IGS symbol name is a CIF user extension which follows a DS command as in 9 PadIn;.

**–i** The CIF file was created by Icarus. An Icarus symbol name is a CIF comment which follows a DS command as in

(9 PadIn);

**–q** The CIF file was created by Squid. A Squid symbol name is a CIF user extension which follows a DS command and specifies the full path name of the file as in

9 /usr/joe/layout/PadIn;

**–m** The CIF file is compatable with the mextra program which uses the symbol rename convention of Berkeley CIF and has the respective layer name in the label user extension as in *94 Text 1500 -9800 NM;.*

**–s** The CIF file was created by Sif. A Sif symbol name is a CIF comment which follows a DS command as in

(Name: PadIn);

**−a** The CIF file was created by Stanford. A Stanford symbol name is a CIF comment which follows a DS command as in
(PadIn);

**−b** The CIF file was created by NCA. A NCA symbol name is a CIF comment which follows a DS command as in
(PadIn);

**−n** The CIF file was created by none of the above.

If the CIF file was not generated by any of the above programs, each CIF symbol will become a STREAM structure with the name *SYMBOLxxx*, where *xxx* is the number of the respective CIF symbol.

*Ciftostrm* will become an interactive program if the command line arguments are insufficient. Input does NOT default to standard I/O. If no command line arguments are given, *ciftostrm* will first prompt you to identify the style of the input CIF file as one of the options mentioned above. The program will next prompt you for the name of the input CIF file. Finally, *Ciftostrm* will prompt you for the name of the CIF-STREAM layer table file.

**SEE ALSO**

strmtocif(CAD1)

**AUTHOR**

Giles Billingsley

## NAME

KIC – Graphics editor for layout of an IC mask set.

## SYNOPSIS

**kic [-gxx] [-d device] [CellNames]**

## DESCRIPTION

*KIC* is an interactive graphics program for laying out IC mask sets. Currently, *KIC* uses the terminal independent graphics package *mfb(3)* and may therefore be used on a variety of graphics terminals. The data model is that of CIF with several enhancements that are described below in the NOTES section.

To learn how to use *KIC*, obtain a copy of the tutorial in ~cad/doc/kic.me. Then log in at one of the several graphics terminals listed below. If you are on an AED, type

kic

or if you are at another graphics terminal, type

kic -gxx

where *xx* is any one of the two character terminal identifiers listed below.

| **ID** | **Terminal Description** |
|---|---|
| A2 | AED 767 with standard release ROM set |
| A8 | AED 512 with standard release ROM set |
| AE | AED 512 with Evans Hall ROM set |
| h0 | hp2648 |
| h1 | hp2648 with three button mouse |
| t2 | Tek 4113 with thumbwheels |
| t3 | Tek 4113 with four button mouse |
| tb | Tek 4105 |
| m1 | Metheus Omega 400 with four button mouse |

If you want *KIC* to send graphics output to a terminal other than the one at which you invoked *KIC*, you can use the **-d device** option. *Device* is the name or full pathname of the respective TTY to receive graphics output from *KIC*.

The program ciftokic translates a CIF file into *KIC* format.

The program kictocif creates a CIF file from a *KIC* cell.

## FILES

~cad/doc/kic.me

## SEE ALSO

ciftokic(CAD1), kictocif(CAD1), KIC tutorial stored in ~cad/doc/kic.me

## AUTHOR

Giles Billingsley
Ken Keller

## BUGS

The known bugs are listed in the *KIC* tutorial.

## NOTES

The wire geometry in *KIC* is assumed to be a square-ended wire that extends a half wire-width beyond its endpoints. In the conversion from *KIC* to CIF, wires are converted to CIF-like wires that are round-ended.

The user extensions that may appear in a *KIC* cell are described below:

**1 ARRAY NumX DX NumY DY;**

> This user extension declares that the next symbol call in the *KIC* cell is to be arrayed. *NumX* is the number of instances in the untransformed array in the positive X direction. *NumY* is the number of instances in the untransformed array in the positive Y direction. The spacing between the bounding boxes of the individual cells of the array in the positive X and Y directions is *DX* and *DY respectively.*

**5 Value String;**

> User extension five defines a property value that is to be assigned to the next primitive in the *KIC* cell. *Value* is a defining integer of the property, and *String* is the character string extension. Property values 1 through 127 and 7000 through 70100 may be created by the strmtokic conversion program and have special meaning to the kictostrm conversion program.

**9 Name;**

> User extension nine defines the name *Name* of the instance to be placed by the next symbol call in the *KIC* cell. If the extension appears before a **DS** command, it redefines the name of the symbol.

**94 Text X Y;**

> This user extension is used to define a text element or label and is considered to be a primitive element as well as a box, wire, polygon, or symbol call. *Text* defines the contents of the label and can not have embedded spaces or control characters. *X, Y* is the lower left coordinate of the label.

## NAME

kictocif – Create a CIF file from a KIC cell.

## SYNOPSIS

**kictocif [-L micperl] [-O ciffile] [-I kicfile] [-T style] [-ads]**

## DESCRIPTION

*Kictocif* translates a hierarchical layout created by *kic(1)* into a CIF file. The **-L** option is used to specify the number of microns per lambda. The **-O** and **-I** options are used to specify the output and input files, respectively.

The **-T** option is used to specify the desired style of CIF. By default, *kictocif* will produce Berkeley CIF. The valid arguments for *style* are as follows:

**k** The CIF style is Berkeley CIF. A symbol name is a CIF user extension which follows a DS command as in *9 PadIn;*. This is the default.

**a** The CIF style is Stanford. A Stanford symbol name is a CIF comment which follows a DS command as in *(PadIn);*.

**b** The CIF style is NCA. A NCA symbol name is a CIF comment which follows a DS command as in *(PadIn);*, and layer names are converted to integers.

**e** The CIF style is Berkeley CIF with property list extensions. A symbol name is a CIF user extension which follows a DS command as in *9 PadIn;*. User extension 5 is used for property lists.

**h** The CIF style is IGS. An IGS symbol name is a CIF user extension which follows a DS command as in *9 PadIn;*.

**i** The CIF style is Icarus. An Icarus symbol name is a CIF comment which follows a DS command as in *(9 PadIn);*.

**m** The CIF is compatable with the mextra program which uses the symbol rename convention of Berkeley CIF and has the respective layer name in the label user extension as in *94 Text 1500 -9800 NM;*.

**s** The CIF style is Sif. A Sif symbol name is a CIF comment which follows a DS command as in *(Name: PadIn);*.

The **-s** option is used if only symbolic layers are to be translated to CIF, and the **-d** option is used if mask layers are to be translated. The **-a** option is used if all layers are to be translated and is the default option.

If the **-O** option is not employed, the name of the output CIF file will be the name of the input *kicfile* appended with *'.cif'*.

*Kictocif* will become an interactive program if the command line arguments are insufficient. Input and output do NOT default to standard I/O. If no command line arguments are given, *kictocif* will first prompt you for the number of microns per lambda. The program will next prompt you for the name of the Root cell of the kic layout. The Root cell is the very top of the hierarchy. Finally, kictocif will ask you if it should convert symbolic layers or only detailed, nonsymbolic layers.

## SEE ALSO

kic(CAD1), ciftokic(CAD1), KIC tutorial stored in ~cad/doc/kic.me

## AUTHOR

Giles Billingsley

## NAME

kictostrm – Create a GDS II STREAM file from a KIC cell hierarchy.

## SYNOPSIS

**kictostrm [-X lfile] [-L micperl] [-Z lname] [-O sfile] [-I kicfile] [-M dbu] [-NC] [-ads]**

## DESCRIPTION

*Kictostrm* translates a hierarchical layout created by *kic(CAD1)* into a GDS II STREAM file. The **-L** option is used to specify the number of microns per lambda. The **-O** and **-I** options are used to specify the output and input files, respectively. If the **-O** option is not used, the name of the output STREAM file will be the name of the input *kicfile* appended with *.str* .

The **-s** option is used if only symbolic layers are to be translated to CIF, and the **-d** option is used if mask layers are to be translated. The **-a** option is used if all layers are to be translated.

The **-M** option is used to specify *dbu* as the number of STREAM database units per micron in the output STREAM file; the default is 100 database units per micron.

The **-X** option is used to specify the file name *lfile* of the CIF-STREAM layer table. The structure of the CIF-STREAM layer table is illustrated below:

```
(number of CIF layer names)
(CIF layer name 1)  (STREAM layer value)  (STREAM datatype value)
(CIF layer name 2)  (STREAM layer value)  (STREAM datatype value)
(CIF layer name 3)  (STREAM layer value)  (STREAM datatype value)
         .
         .
         .
        etc.
```

The STREAM layer and datatype values must be non-negative integers, and the CIF layer names should be unique. Comments may be appended to the CIF-STREAM layer table after the last CIF layer definition.

The name of the STREAM library file may be specified to be *lname* with the **-Z** option. The default library name is *KICTOSTREAM* .

If the **-N** option is used, *kictostrm* will expect layer names such as those produced by default by the *strmtokic* program. These layer names will consist of four digits; the first two digits represent the STREAM layer value, and the next two digits represent the STREAM datatype. For example, the layer *1234* would represent STREAM layer 12 and STREAM datatype 34.

It may be undesirable to have symbol definitions of standard library cells placed in the output STREAM file. If the **-C** option is used, *kictostrm* will place a symbol definition in the output STREAM file if and only if the symbol exists in the current working directory.

*Kictostrm* will become an interactive program if the command line arguments are insufficient. Input and output do NOT default to standard I/O. If no command line arguments are given, *kictostrm* will first prompt you for the number of microns per lambda. The program will next prompt you for the name of the Root cell of the kic layout. The Root cell is the very top of the hierarchy. Finally, kictostrm will ask you if it should convert symbolic layers, detailed, non-symbolic layers, or all layers.

## SEE ALSO

kic(CAD1), strmtokic(CAD1), strmlyrtbl(CAD1),
KIC tutorial stored in ~cad/doc/kic.me

## AUTHOR

Giles Billingsley

## NOTES

*Kictostrm* will accept several property extensions for the purpose of modifying the output STREAM file. The following extensions are relevant only to a symbol definition and must physically appear before the **DS** command in the KIC cell.

**5 7000 num;**

This property extension defines *num* as the Calma GDS II version number of the output STREAM file.

**5 7002 name;**

The property value 7002 defines the property string *name* as the name of the STREAM library.

**5 7032 font1 font2 font3 font4;**

The property value 7032 declares that the property string contains from zero to four STREAM textfont file names. These file names are meaningful only to the GDS II system.

**5 7034 generations;**

This property extension defines *generations* as the number of generations or previous versions to be maintained by GDS II. This number is meaningful only to the GDS II system.

**5 7035 attribute_file;**

The property value 7035 declares the the property string *attribute_file* contains the name of the GDS II attributes file. This file name is meaningful only to the GDS II system.

The following extensions are relevant to geometries and would physically appear before the respective object in the KIC cell.

**5 7033 PATHTYPE type;**

The property 7033 is used to define the type of a wire. If the integer *type* is zero, the next wire defined in the KIC cell will have square ends that are flush with the endpoint. If *type* is one, the wire will have rounded ends. By default, wires will be pathtype two which are square-ended and extend one half wire-width beyond the endpoints.

**5 7012 WIDTH w PRESENT p PTYPE t MAG m ANGLE a REFLECT r;**

Property 7012 will always precede the KIC label extension number 94, and is used to modify the label declaration. The integer *w* defines the line width of the label in lambda units, *p* is the presentation bit mask, and *t* is the pathtype as defined above.

**5 value string;**

This extension is used to attach a STREAM attribute to an object. The integer *value* is between 1 and 127, and *string* is a character string.

## NAME

scale – Scale a KIC cell hierarchy.

## SYNOPSIS

**scale [-a numerator] [-b denominator] [RootFile]**

## DESCRIPTION

*Scale* will change the resolution of a hierarchical layout created by *kic(1)*. The scaling factor is a ratio of two integers that are specified on the command line and that default to unity. The **-a** option is used to specify the numerator of the scaling factor, and the **-b** option is used to specify the denominator of the scaling factor. *RootFile* is the filename of the top (root) cell in the layout hierarchy.

*Scale* will become an interactive program if the command line arguments are insufficient. Input and output do NOT default to standard I/O. If no command line argument is given for the top of the cell hierarchy, *scale* will prompt you for the file name.

## SEE ALSO

kic(CAD1), KIC tutorial stored in ~cad/doc/kic.me

## AUTHOR

Giles Billingsley

## BUGS

Integer overflows may occur if the scaling factor exceeds 100.

**NAME**

strmlyrtbl - make CIF-STREAM layer table from STREAM file

**SYNOPSIS**

strmlyrtbl [-id] tablefile streamfile

**DESCRIPTION**

*strmlyrtbl* is an interactive program that will determine all layers and corresponding datatypes in a STREAM file *streamfile* and will request from the user through the standard input the CIF layer name for each layer-datatype pair. From this information, *strmlyrtbl* will create a CIF-STREAM layer table *tablefile*.

The structure of the CIF-STREAM layer table is illustrated below:

```
(number of CIF layer definitions)
(CIF layer name 1)  (STREAM layer value)  (STREAM datatype value)
(CIF layer name 2)  (STREAM layer value)  (STREAM datatype value)
(CIF layer name 3)  (STREAM layer value)  (STREAM datatype value)
    .
    .
    .
   etc.
```

The STREAM layer and datatype values must be non-negative integers, and the CIF layer names should be unique. Comments may be appended to the CIF-STREAM layer table after the last CIF layer definition.

After creating th CIF-STREAM layer table *strmlyrtbl* will also generate a file called **.KIC** in the current directory; this **.KIC** file can be used with the conversion program *kictostrm* or the layout editor *kic*. Note that the CIF layer name and the corresponding KIC layer name are identical. See kic(CAD1) and kictostrm(CAD1).

**AUTHOR**

Giles Billingsley

**SEE ALSO**

kictostrm(CAD1), strmtokic(CAD1), ciftostrm(CAD1), strmtocif(CAD1)

## NAME

strmtocif – Create a GDS II STREAM file from a CIF file.

## SYNOPSIS

**strmtocif [-BE] [-Xltable] [-Csname] [ streamfile [ciffile]]**

## DESCRIPTION

*Strmtocif* translates a GDS II STREAM file into a Berkeley style CIF file.

If the **-B** option is used, polygons with manhattan edges (i.e., all angles of the contour are integer multiples of 90 degrees) will be decomposed into boxes. If the **-E** option is present, all errors and diagnostic messages will be printed in a file named *strmtocif.err* rather than at the standard error output device.

The **-C** option is used to specify *sname* as the name of the symbol that is at the top of the STREAM hierarchy. This symbol is called outside of all symbol definitions in the output CIF file, and the default is the first symbol that appears in the STREAM file.

The input STREAM file can be specified on the command line as *streamfile*. The default input is *stdin*. The output CIF file can be specified on the command line as *ciffile* only if the input STREAM file has been specified. The default output is *stdout*.

The **-X** option is used to specify the file name *lfile* of the CIF-STREAM layer table. The structure of the CIF-STREAM layer table is illustrated below:

```
(number of CIF layer names)
(CIF layer name 1)   (STREAM layer value)   (STREAM datatype value)
(CIF layer name 2)   (STREAM layer value)   (STREAM datatype value)
(CIF layer name 3)   (STREAM layer value)   (STREAM datatype value)
          .
          .
          .
         etc.
```

The STREAM layer and datatype values must be non-negative integers, and the CIF layer names should be unique. Comments may be appended to the CIF-STREAM layer table after the last CIF layer definition.

If no CIF-STREAM layer table is specified, the CIF layer names will be printed as strings of four digits; the first two digits represent the STREAM layer value, and the next two digits represent the STREAM datatype. For example, the CIF layer *1234* would represent STREAM layer 12 and STREAM datatype 34.

## SEE ALSO

ciftostrm(CAD1)

## AUTHOR

Giles Billingsley

## NAME

strmtokic – Translate a GDS II STREAM file into KIC format.

## SYNOPSIS

**strmtokic [-L micperl] [-BE] [-C sname] [-R rootfile] [-X lfile] [streamfile]**

## DESCRIPTION

*Strmtokic* translates a GDS II STREAM file into the hierarchical directory format readable by *KIC(1)*. The **-L** option is used to specify the number of microns per lambda. If the **-B** option is used, polygons with manhattan edges (i.e., all angles of the contour are integer multiples of 90 degrees) will be decomposed into boxes. If the **-E** option is present, all errors and diagnostic messages will be printed in a file named *strmtokic.err* rather than at the standard error output device.

The **-R** option is used to specify the name of the Root cell. The Root cell is the top of the hierarchy, and the default name is *Root*. The **-C** option is used to specify *sname* as the name of the symbol that is at the top of the STREAM hierarchy. This symbol is called by the Root cell, and the default is the first symbol that appears in the STREAM file.

The input STREAM file can be specified on the command line as *streamfile*. The default input is *stdin*.

The **-X** option is used to specify the file name *lfile* of the CIF-STREAM layer table. The structure of the CIF-STREAM layer table is illustrated below:

```
(number of CIF layer names)
(CIF layer name 1)   (STREAM layer value)   (STREAM datatype value)
(CIF layer name 2)   (STREAM layer value)   (STREAM datatype value)
(CIF layer name 3)   (STREAM layer value)   (STREAM datatype value)
          .
          .
          .
          etc.
```

The STREAM layer and datatype values must be non-negative integers, and the CIF layer names should be unique. Comments may be appended to the CIF-STREAM layer table after the last CIF layer definition. The CIF-STREAM layer table can be created by the *strmlyrtbl* program.

If no CIF-STREAM layer table is specified, the CIF layer names will be printed as strings of four digits; the first two digits represent the STREAM layer value, and the next two digits represent the STREAM datatype. For example, the CIF layer *1234* would represent STREAM layer 12 and STREAM datatype 34.

## SEE ALSO

kic(CAD1), kictostrm(CAD1), strmlyrtbl(CAD1),
Kic tutorial stored in ~cad/doc/kic.me

## AUTHOR

Giles Billingsley

## BUGS

Text labels in KIC are not allowed to have embedded space characters. *Strmtokic* will convert all spaces in STREAM text elements to underscores.

At the present time, KIC has no notion of symbol magnification. All STREAM symbol or structure calls must therefore have a unity magnification factor. *Strmtokic* will warn the user if it encounters a symbol call with a non-unity magnification.

## NOTES

*Strmtokic* will generate several property extensions in the KIC cells that are meaningful to the *strmtokic* program. The following KIC extensions are relevant to a symbol definition and would physically appear in the KIC cell before the **DS** command.

**5 7000 num;**

This property user extension defines *num* as the Calma GDS II version number of the STREAM file from which the KIC cell was extracted.

**5 7002 name;**

The property value 7002 defines the property string *name* as the name of the stream library from which the KIC cell was extracted.

**5 7032 font1 font2 font3 font4;**

The property value 7032 declares that the property string contains from zero to four STREAM textfont file names. These file names are meaningful only to the GDS II system.

**5 7034 generations;**

This property extension defines *generations* as the number of generations or previous versions to be maintained by GDS II. This number is meaningful only to the GDS II system.

**5 7035 attribute_file;**

The property value 7035 declares the the property string *attribute_file* contains the name of the GDS II attributes file. This file name is meaningful only to the GDS II system.

The following extensions are relevant to geometries and would physically appear before the respective object in the KIC cell.

**5 7033 PATHTYPE type;**

The property 7033 is used to define the type of a wire. If the integer *type* is zero, the next wire defined in the KIC cell will have square ends that are flush with the endpoint. If *type* is one, the wire will have rounded ends. By default, wires will be pathtype two which are square-ended and extend one half wire-width beyond the endpoints.

**5 7012 WIDTH w PRESENT p PTYPE t MAG m ANGLE a REFLECT r;**

Property 7012 will always precede the KIC label extension number 94, and is used to modify the label declaration. The integer *w* defines the line width of the label in lambda units, *p* is the presentation bit mask, and *t* is the pathtype as defined above.

**5 value string;**

This extension is used to attach a STREAM attribute to an object. The integer *value* is between 1 and 127, and *string* is a character string.

# References

[1] K. H. Keller and A. R Newton, *KIC2: A Low-Cost Interactive Editor for Integrated Circuit Design*, Digest of Papers, IEEE Compcon 82 Conf., San Fransisco, California, February 22-25, 1982, pp. 302-304.

[2] K. H. Keller, *KIC, A Graphics Editor for Integrated Circuits*, Masters Report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Ca., June 1981.

[3] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[4] C. A. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass., 1980.

[5] R. Hon ad C. Sequin, *A Guide to LSI Implementation*, Xerox PARC Technical Report SSL-79-7, 1980.

[6] W. M. Newman and R. F. Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill, New York, 1977.

[7] J. D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass., 1982.

[8] M. H. Arnold and J. K. Ousterhout, *LYRA: A New Approach to Geometric Layout Rule Checking*, Proc. 19th ACM IEE Design Automation Conf. Las Vegas, Nevada, June 14-16, 1982, pp.530-536.

[9] *Programming in VAX-11C*, Digital Equipment Corporation, P.O. Box CS2008, Nashua, New Hampshire. 03061

[10] KIC2 under VAX/VMS is also distributed by the Engineering Systems Group of Digital Equipment Corp., 2 Iron Way, Marlboro, Mass. 01752

[11] STREAM is a proprietary description format for graphic data that is licensed by Calma, Inc. The conversion programs between KIC and Calma STREAM were developed at Tektronix, Inc., and the Electronics Research Laboratory with the permission of Calma and may be released by the Electronics Research Laboratory only to licensed customers of Calma.