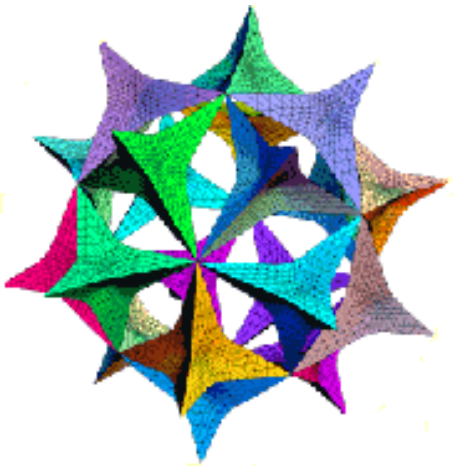


GDSIIExporter

(Version 2.0)

Context

Thank you for interest in ***GDSIIExporter***.

This document does not pretend to be a manual on the GDSII stream format or some kind training course. The document is useful for users of ***GDSIIExporter*** (component) to build their own projects with GDSII format exporting feature.
To get the complete manual of the format and sample files, please visit http://boolean.klaasholwerda.nl/interface/bnf/gdsformat.html. Also, see web site of Cadense Systems Inc., http://www.cadence.com.

Introduction

The GDSII stream format is a standard file format for transferring or archiving 2D graphical design data. The GDSII is most commonly used format for electron beam lithography, photo mask production and optical circuit design.

GDSIIExporter was implemented as dynamic link library. It allows using ***it*** with any language or system that supports calls to dll.

Try DEMO version of ***GDSIIExporter*** and check it out if it's useful for your project.
Please, review my web site www.EquTranslator.com regularly for new products and services.
For more detail information, please contact me at:
support@EquTranslator.com

Interface

The interface was designed with respect to well-known Buchus Naur Form, which is logical representation of the GDSII format.
First of all let's see what Buchus Naur Form is. The form uses the following symbols:

Symbol Name	Symbol	Meaning
Double Colon	::	"Is composed of."
Square brackets	[ ]	An element which can occur zero or one time.
Braces	{ }	Choose one of the elements within the braces.
Braces with an asterisk	{ }*	The elements within the braces can occur zero or more times.
Braces with a plus	{ }+	The elements within braces must occur one or more times.
Angle brackets	< >	These elements are further defined as separate entities in the syntax list.
Vertical bar	|	Or

The following is the Bachus Naur Form of the GDSII format; the words in capital are the names of *RECORDS*

<stream format> ::=	HEADER BGNLIB [LIBDIRSIZE] [SRFNAME] [libsecur] libname [reflibs] [fonts] [attrtable] [generations] [<FormatType>] UNITS {<structure>}* ENDLIB
<FormatType> ::=	FORMAT | FORMAT {MASK}+ ENDMASKS
<structure> ::=	BNGSTR STRNAME [STRCLASS] {<element>}* ENDSTR
<element> ::=	{<boundary> | <path> | <SREF> | <AREF> | <text> | <node> | <box>} {<property>}* ENDEL
<boundary> ::=	BOUNDARY [EFLAGS] [PLEX] LAYER DATATYPE XY
<path> ::=	PATH [EFLAGS] [PLEX] LAYER DATATYPE [PATHTYPE] [WIDTH] [BGNEXTN] [ENDEXTN] XY
<SREF> ::=	SREF [EFLAGS] [PLEX] SNAME [<strans>] XY
<AREF> ::=	AREF [EFLAGS] [PLEX] SNAME [<strans>] COLROW XY
<text> ::=	TEXT [EFLAGS] [PLEX] LAYER <textbody>
<node> ::=	NODE [EFLAGS] [PLEX] LAYER NODETYPE XY
<box> ::=	BOX [EFLAGS] [PLEX] LAYER BOXTYPE XY
<textbody> ::=	TEXTTYPE [PRESENTATION] [PATHTYPE] [WIDTH] [<strans>] XY STRING
<strans> ::=	STRANS [MAG] [ANGLE]
<property> ::=	PROPATTR PROPVALUE

The interface corresponds to Bachus Naur Form that helps better understand and use ***GDSIIExporter***.

The GDSII format is a binary format that is platform independent, because it uses internally defined formats and types (real, integers, string length, date and time records etc.). So, it is very important to use right type of the function arguments to send a data to ***GDSIIExporter***.

To describe the interface function signature, here are used C/C++ basic types that can be represented by equivalent types in other languages and systems. The types **short (__int16)**, **int (__int32)**, **double** have length 2 byte, 4 byte, 8 byte correspondently and **char*** represents the C/C++ null terminated string.

Each interface function returns integer value to indicate success of execution. In GDSIIConst.h file you can find list of error messages and return codes. For more information see: Error Handling. Examples.

Also, each interface function has input parameter:
HANDLE hnd
that uniquely identify current session.

```
int GDS_BeginLib(const char *fileName, double dbUserUnit,
                 double dbUnitInMeter, HANDLE *hnd);

int  GDS_EndLib(HANDLE hnd);
```

The functions should be used when GDSII format file are started and ended. This is logical "brackets" and must be called only once for each new GDSII file.
The functions create records, which can be represented in Bachus Naur Form as:

```
HEADER BGNLIB LIBNAME UNITS
[{BGNSTR STRNAME [{<element>}+] ENDSTR}+]
ENDLIB
```

`[{BGNSTR STRNAME [{<element>}+] ENDSTR}+]` – the structures and elements. An element portion of a stream file: <boundary> | <path> | <sref> | <aref> | <text> | <node> | <box> ENDEL. The structures and elements can be inserted in the file by other interface functions (see below).

The arghuments:
filename – name of the GDSII format file;
dbUserUnit - the size of a database unit in user units;
dbUnitInMeter - the size of a database unit in meters.
hnd – handle to the current session. Calling GDS_BeginLib the value of session must be initialized.

For example, if you create a library with the default units (user unit = 1 micron and 1000 database units per user unit), the first number is .001, and the second number is 1E-9. Typically, the first number is less than 1, since you use more than 1 database unit per user unit. To calculate the size of a user unit in meters, divide the second number by the first.

Example:

```
#include "GDSIIInterface.h"
#include "GDSIIConst.h"

#pragma comment(lib, "GDSIIExporter.lib")

int main(int argc, char* argv[])
{
        HANDLE hnd1;
        HANDLE hnd2;

        char* f1="tes1.gds";
        char* f2="test2.gds";

        GDS_BeginLib(f1, 1e-3, 1e-9, &hnd1);
                // [{BGNSTR STRNAME [{<element>}+] ENDSTR}+]
        GDS_EndLib(hnd1);

        GDS_BeginLib(f2, 1e-3, 1e-9, &hnd2);
                // [{BGNSTR STRNAME [{<element>}+] ENDSTR}+]
        GDS_EndLib(hnd2);
        return 0;
}
```

int GDS_BeginStructure(HANDLE hnd, const char* strName);

int GDS_EndStructure(HANDLE hnd);

The functions are logical “brackets” for structure element. The GDSII format presumes zero or more structures in the library. In Bachus Naur Form the element can be represented as:

BGNSTR STRNAME [STRCLASS] [{<element>}+] ENDSTR

Each structure has two header records and one tail record that sandwich an arbitrary list of elements. The first structure header is the BGNSTR record, which contains the creation date and the last modification date. Following that is the STRNAME record, which names the structure using any alphabetic or numeric characters, the dollar sign, or the underscore. Then the structure is opened and any of the elements can be listed. The last record of the structure is ENDSTR. Following it must be another BGNSTR or the end of the library, ENDLIB.

The arguments:
hnd – handle of current session;
strName – name of the structure. Up to 32 characters in GDSII, A-Z, a-z, 0-9, _, ?, and $ are all legal characters.
The functions create records, which can be represented in Bachus Naur Form as:

```
BGNSTR STRNAME {<element>}* ENDSTR
```

*{<element>}** - element portion of a stream file: <boundary> | <path> | <sref> | <aref> | <text> | <node> | <box> ENDEL.

Example:

```
#include "GDSIIInterface.h"
#include "GDSIIConst.h"

#pragma comment(lib, "GDSIIExporter.lib")

int main(int argc, char* argv[])
{
        HANDLE hnd;
        char* f="test.gds";

        GDS_BeginLib(f, 1e-3, 1e-9, &hnd);

                GDS_BeginStructure(hnd, "str1");
                        // {<element>}*
                GDS_EndStructure(hnd);

                GDS_BeginStructure(hnd, "str2");
                        // {<element>}*
                GDS_EndStructure(hnd);

        GDS_EndLib(hnd);

        return 0;
}
```

void GDS_Path(HANDLE hnd , int *pointsX, int *pointsY, short pointsNum, int width, short layer, short datatype, short pathtype, short elflags, int plex);

The function creates a PATH element. In Bachus Naur Form the element can be represented as:

```
PATH [ELFLAGS][PLEX] LAYER DATATYPE [PATHTYPE]
[WIDTH] XY
```

A path is an open figure with a nonzero width that is typically used to place wires. This element is initiated with a PATH record followed by the optional ELFLAGS and PLEX records. The LAYER record must follow to identify the desired path material. Also, a DATATYPE record must appear and an XY record to define the coordinates of the path. From two to 200 points may be given in a path.
Prior to the XY record of a path specification there may be two optional records called PATHTYPE and WIDTH. The PATHTYPE record describes the nature of the path segment ends, according to its parameter value. If the value is 0, the segments will have square ends that terminate at the path vertices. The value 1 indicates rounded ends and the value 2 indicates square ends that overlap their vertices by one-half of their width. The width of the path is defined by the optional WIDTH record.

The arguments:
hnd – handle of current session;
pointsX, *pointsY* – arrays of the XY coordinates to be connected by a segment;
pointsNum – number of points in the arrays *pointsX* and *pointsY*. Path elements may have a minimum of 2 and a maximum of 200

coordinates. The max points can be changed up on customer request;

width - [WIDTH], specifies the width of a path or text lines in database units. A negative value for width means that the width is absolute, that is, the width is not affected by the magnification factor of any parent reference. If omitted, zero is assumed;

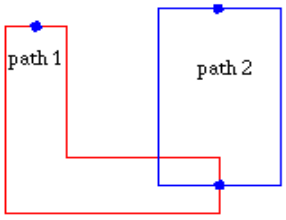
layer – LAYER, the value of the layer must be in the range of 0 to 255;

datatype – DATATYPE, the value of the datatype must be in the range of 0 to 255;

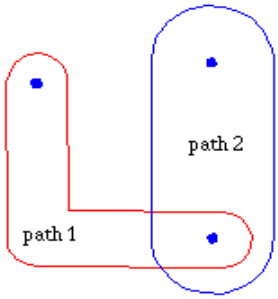
pathtype - [PATHTYPE], value that describes the type of path endpoints. The value is

- 0 for square-ended paths that end flush with their endpoints
- 1 for round-ended paths
- 2 for square-ended paths that extend a half-width beyond their endpoints

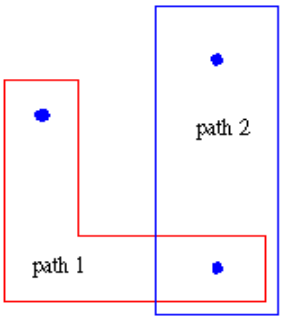
If not specified, a Path-type of 0 is assumed. The following picture shows the path types:



Pathtype 0 produces a square-ended path, ending flush with the digitized endpoints. This is the default pathtype if none is specified



Pathtype 1 produces a round-ended path. The two ends are semicircular with center at the digitized endpoints.



Pathtype 2 produces a square-ended path. The ends of the pathextend beyond the digitized end-points by one-half the path width.

Predefined constants (see GDSIIConst.h):

```
// PATHTYPE
const   __int16 PATHTYPE0=0;
const   __int16 PATHTYPE1=1;
const   __int16 PATHTYPE2=2;
```

elflags - [ELFLAGS], bit 15 (the rightmost bit)specifies Template data. Bit 14 specifies External data (also referred to as Exterior data). All other bits are currently unused and must be cleared to 0. If this record is omitted, all bits are assumed to be 0. The following shows an ELFLAGS record. For additional information on Template data, consult the GDSII Reference Manual.
Predefined constants:

```
//ELFLAGS
const   __int16 ELFLAGS_0=0;
const   __int16 ELFLAGS_TEMPLATE=1;
const   __int16 ELFLAGS_EXTERIOR=2;
```

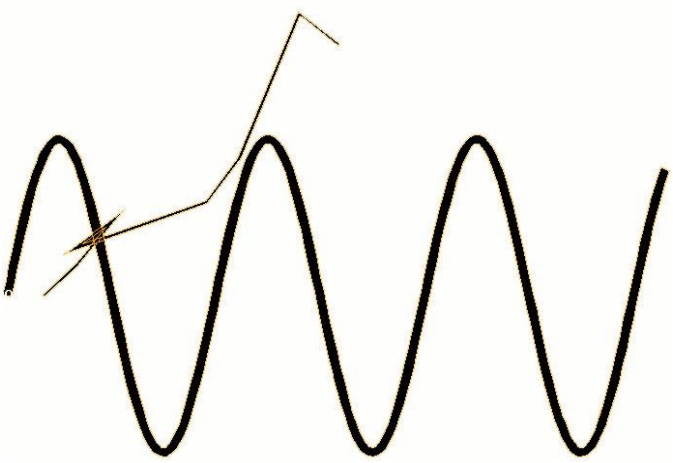
plex - [PLEX], a unique positive number which is common to all elements of the plex to which this element belongs. The head of the plex is flagged by setting the seventh bit; therefore, plex numbers should be small enough to occupy only the right-most 24 bits. Predefined constant:

```
const  __int32 PLEX_HEAD=0x1000000
```

Example:

```
#include <math.h>
#include "GDSIIInterface.h"
#include "GDSIIConst.h"

#pragma comment(lib, "GDSIIExporter.lib")

#define pt 199

int main(int argc, char* argv[])
{
        HANDLE hnd;
        char* f="test.gds";
        int x1[pt];
        int y1[pt];

        int x[]={110,210,327,200,600,700,880,1000};
        int y[]={0,100,250,150,300,440,900,800};

        //To build SIN function
        double k;
        for(int i=0; i<pt; i++)
        {
                k=i/10.;
                x1[i]=i*10;
                y1[i]=long(sin(k)*500);
        }

        GDS_BeginLib(f, 1e-3, 1e-9, &hnd);

         GDS_BeginStructure(hnd, "str1");
                GDS_Path(hnd, x, y, 8 ,10, 1,63, PATHTYPE1, 0, 0);
                GDS_Path(hnd, x1, y1, pt , 30, 1, 63, PATHTYPE1,
ELFLAGS_TEMPLATE,0);
         GDS_EndStructure(hnd);

        GDS_EndLib(hnd);

        return 0;
}
```

Result:



void GDS_Arc(HANDLE hnd, int centerX, int centerY, int radius, double startAng, double endAng, short segmentNum, int width, short layer, short datatype, short pathtype, short elflags, int plex);

The function is an extension of the GDSII stream format and builds an arc. GDS_Arc uses PATH record and as result uses similar arguments (see GDS_Path).
The arguments:
hnd – handle of current session;
centerX, *centerY* –coordinates of an arc center;
radius – radius of an arc;
startAng, *endAng* - start angle and end angle of an arc in degree;
segmentNum – a number of segments to draw an arc. The number is limited by 200 (see max pints for PATH).
width - [WIDTH], specifies the width of a path or text lines in database units. A negative value for width means that the width is absolute, that is, the width is not affected by the magnification factor of any parent reference. If omitted, zero is assumed;
layer – LAYER, the value of the layer must be in the range of 0 to 255;
datatype – DATATYPE, the value of the datatype must be in the range of 0 to 255;
pathtype - [PATHTYPE], value that describes the type of path endpoints. The value is

- 0 for square-ended paths that end flush with their endpoints
- 1 for round-ended paths
- 2 for square-ended paths that extend a half-width beyond their endpoints

If not specified, a Path-type of 0 is assumed (see GDS_Path for more details).
Predefined constants:

```
// PATHTYPE
const   __int16 PATHTYPE0=0;
const   __int16 PATHTYPE1=1;
const   __int16 PATHTYPE2=2;
```

elflags - [ELFLAGS], bit 15 (the rightmost bit)specifies Template data. Bit 14 specifies External data (also referred to as Exterior data). All other bits are currently unused and must be cleared to 0. If this record is omitted, all bits are assumed to be 0. The following shows an ELFLAGS record. For additional information on Template data, consult the GDSII Reference Manual.
Predefined constants:

```
//ELFLAGS
const   __int16 ELFLAGS_0=0;
const   __int16 ELFLAGS_TEMPLATE=1;
const   __int16 ELFLAGS_EXTERIOR=2;
```

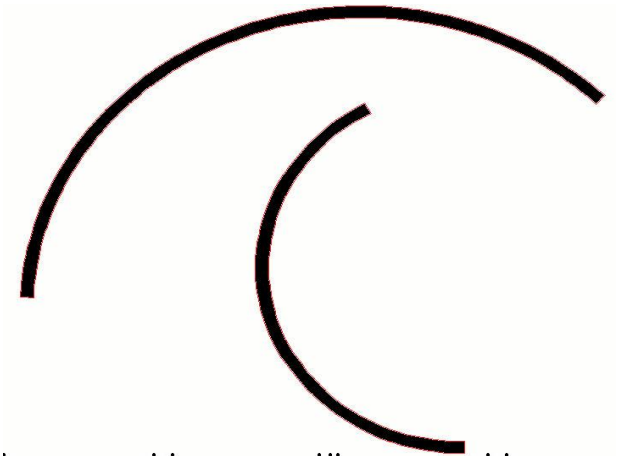
plex - [PLEX], a unique positive number which is common to all elements of the plex to which this element belongs. The head of the plex is flagged by setting the seventh bit; therefore, plex numbers should be small enough to occupy only the right-most 24 bits.
Predefined constant:

```
const  __int32 PLEX_HEAD=0x1000000
```

Example:

```
#include "GDSIIInterface.h"
#include "GDSIIConst.h"

#pragma comment(lib, "GDSIIExporter.lib")

int main(int argc, char* argv[])
{
        HANDLE hnd;
        char* f="test.gds";

        GDS_BeginLib(f, 1e-3, 1e-9, &hnd);

                GDS_BeginStructure(hnd, "str1");
                        GDS_Arc(hnd, 100, 150, 1000, 45, 178, 135,40,
0, 0, PATHTYPE1,0,0);
                        GDS_Arc(hnd, 400, 300, 600, 120, 270, 100,40,
0, 0, PATHTYPE2,0,0);

                GDS_EndStructure(hnd);

        GDS_EndLib(hnd);

        return 0;
}
```

Result:



void GDS_Boundary(HANDLE hnd, int *pointsX, int *pointsY, short pointsNum, short layer, short datatype, short elflags, int plex);

The function creates a BOUDARY element. In Bachus Naur Form the element can be represented as:

```
BOUNDARY[ELFLAGS][PLEX]LAYER DATATYPE XY
```

The boundary element defines a filled polygon. It begins with a BOUNDARY record, has an optional ELFLAGS and PLEX record, and then has required LAYER, DATATYPE, and XY records.
The ELFLAGS record, which appears optionally in every element, has two flags in its parameter to indicate template data (if bit 16 is set) or external data (if bit 15 is set). This record should be ignored on input and excluded from output. Note that the GDS II integer has bit 1 in the leftmost or most significant position so these two flags are in the least significant bits.
The PLEX record is also optional to every element and defines element structuring by aggregating those that have common plex numbers. Although a 4-byte integer is available for plex numbering, the high byte (first byte) is a flag that indicates the head of the plex if its least significant bit (bit 8) is set.
The LAYER record is required to define which layer is to be used for this boundary. The meaning of these layers is not defined rigorously and must be determined for each design environment and library.
The DATATYPE record contains unimportant information and its argument can be zero.
The XY record contains anywhere from four to 200 coordinate pairs that define the outline of the polygon. The record length defines the number of points in this record. Note that boundaries must be closed explicitly, so the first and last coordinate values must be the same.

The arguments:
hnd – handle of current session;
pointsX, *pointsY* – arrays of the XY coordinates to be connected by a segment;
pointsNum – number of points in the arrays *pointsX* and *pointsY*. Boundary elements may have a minimum of 4 and a maximum of 200 coordinates. The max points can be changed up on customer request;
layer – LAYER, the value of the layer must be in the range of 0 to 255;
datatype – `DATATYPE, t`he value of the datatype must be in the range of 0 to 255;
elflags - [ELFLAGS], bit 15 (the rightmost bit)specifies Template data. Bit 14 specifies External data (also referred to as Exterior data). All other bits are currently unused and must be cleared to 0. If this record is omitted, all bits are assumed to be 0. The following shows an ELFLAGS record. For additional information on Template data, consult the GDSII Reference Manual.
Predefined constants:

```
//ELFLAGS
const   __int16 ELFLAGS_0=0;
const   __int16 ELFLAGS_TEMPLATE=1;
const   __int16 ELFLAGS_EXTERIOR=2;
```

plex - [PLEX], an unique positive number which is common to all elements of the plex to which this element belongs. The head of the plex is flagged by setting the seventh bit; therefore, plex numbers should be small enough to occupy only the right-most 24 bits. Predefined constant:

```
const  __int32 PLEX_HEAD=0x1000000
```

Example:

```
#include "GDSIIInterface.h"
#include "GDSIIConst.h"

#pragma comment(lib, "GDSIIExporter.lib")

int main(int argc, char* argv[])
{
        HANDLE hnd;
        char* f="test.gds";

        int x[]={110,210,327,550};
        int y[]={0,200,250,170};

        GDS_BeginLib(f, 1e-3, 1e-9, &hnd);
                GDS_BeginStructure(hnd, "str1");
                        GDS_Boundary(hnd, x, y, 4 ,1, 61, 0, 0);
                GDS_EndStructure(hnd);
        GDS_EndLib(hnd);

        return 0;
}
```

Result:

void GDS_Circle(HANDLE hnd, int centerX, int centerY, int radius, short segmentNum, short layer, short datatype, short elflags, int plex);

The function is an extension of the GDSII stream format and builds a circle. GDS_Circle uses BOUNDORY record and as result uses similar arguments (see GDS_Boundary).
The arguments:
hnd – handle of current session;
centerX, *centerY* –coordinates of a circle center;
radius – radius of a circle;
segmentNum – a number of segments to draw a circle. The number is limited by 200 (see max pints for BOUNDARY).
layer – LAYER, the value of the layer must be in the range of 0 to 255;
datatype – `DATATYPE, t`he value of the datatype must be in the range of 0 to 255;
elflags - [ELFLAGS], bit 15 (the rightmost bit)specifies Template data. Bit 14 specifies External data (also referred to as Exterior data). All other bits are currently unused and must be cleared to 0. If this record is omitted, all bits are assumed to be 0. The following shows an ELFLAGS record. For additional information on Template data, consult the GDSII Reference Manual.
Predefined constants:

```
//ELFLAGS
const   __int16 ELFLAGS_0=0;
const   __int16 ELFLAGS_TEMPLATE=1;
const   __int16 ELFLAGS_EXTERIOR=2;
```

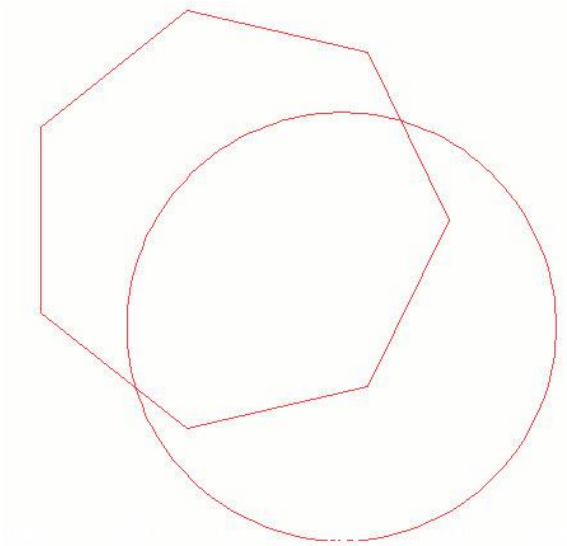
plex - [PLEX], an unique positive number which is common to all elements of the plex to which this element belongs. The head of the plex is flagged by setting the seventh bit; therefore, plex numbers should be small enough to occupy only the right-most

24 bits. Predefined constant:
const __int32 PLEX_HEAD=0x1000000

Example:

```
#include "GDSIIInterface.h"
#include "GDSIIConst.h"

#pragma comment(lib, "GDSIIExporter.lib")

int main(int argc, char* argv[])
{
        HANDLE hnd;
        char* f="test.gds";

        GDS_BeginLib(f, 1e-3, 1e-9, &hnd);
                GDS_BeginStructure(hnd, "str1");
                        GDS_Circle(hnd, 1500, 1000, 1000, 150,
0,0,0,0);
                        GDS_Circle(hnd, 1000, 1500, 1000, 7, 0,0,0,0);
                GDS_EndStructure(hnd);
        GDS_EndLib(hnd);

        return 0;
}
```

Result:



void GDS_Sector(HANDLE hnd, int centerX, int centerY,
int radius, double startAng, double endAng,
short segmentNum, short layer,
short datatype, short elflags, int plex);

The function is an extension of the GDSII stream format and builds a sector. GDS_Sector uses BOUNDORY.
The arguments:
hnd – handle of current session;
centerX, *centerY* –coordinates of the sector center;
radius – radius of a sector;
startAng, *endAng* - start angle and end angle of an arc in degree;
segmentNum – a number of segments to draw a circle. The number is limited by 200 (see max pints for BOUNDARY).
layer – LAYER, the value of the layer must be in the range of 0 to 255;
datatype – DATATYPE, the value of the datatype must be in the range of 0 to 255;
elflags - [ELFLAGS], bit 15 (the rightmost bit)specifies Template data. Bit 14 specifies External data (also referred to as Exterior data). All other bits are currently unused and must be cleared to 0. If this record is omitted, all bits are assumed to be 0. The following shows an ELFLAGS record. For additional information on Template data, consult the GDSII Reference Manual.
Predefined constants:

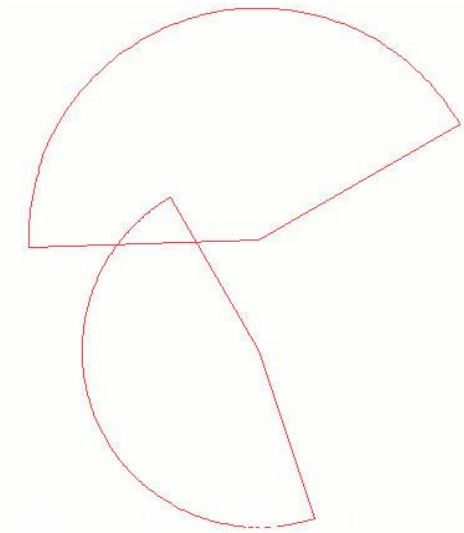
```
//ELFLAGS
const   __int16 ELFLAGS_0=0;
const   __int16 ELFLAGS_TEMPLATE=1;
const   __int16 ELFLAGS_EXTERIOR=2;
```

plex - [PLEX], an unique positive number which is common to all elements of the plex to which this element belongs. The head of the plex is flagged by setting the seventh bit; therefore, plex numbers should be small enough to occupy only the right-most 24 bits. Predefined constant:
const __int32 PLEX_HEAD=0x1000000

Example:

```
#include "GDSIIInterface.h"
#include "GDSIIConst.h"

#pragma comment(lib, "GDSIIExporter.lib")

int main(int argc, char* argv[])
{
        HANDLE hnd;
        char* f="test.gds";

        GDS_BeginLib(f, 1e-3, 1e-9, &hnd);
            GDS_BeginStructure(hnd, "str1");
                GDS_Sector(hnd, 1000, 1500, 1050, 30, 185,100, 0,0,0,0);
                GDS_Sector(hnd, 1000, 1000, 800, 120, 290,199, 0,0,0,0);
            GDS_EndStructure(hnd);
        GDS_EndLib(hnd);

        return 0;
}
```

Result:



```
void  GDS_Box(HANDLE hnd, int x1, int y1, int x2, int y2,
                short layer, short boxtype,
                short elflags, int plex);

void  GDS_BoxWH(HANDLE hnd, int x, int y, int width,
                int height,          short layer,
                 short boxtype,  short elflags,
                int plex);

void GDS_Square(HANDLE hnd, int x, int y, int side,
                short layer, short boxtype,  short elflags, int
                plex);
```

The function creates a BOX element. In Bachus Naur Form the element can be represented as:

```
BOX [ELFLAGS] [PLEX] LAYER BOXTYPE XY
```

Optional ELFLAGS and PLEX records follow the BOX record, a mandatory LAYER record, a BOXTYPE record with a zero argument, and an XY record. The XY must contain five points that describe a closed, four-sided box. Unlike the boundary, this is not a filled figure.

The arguments:
hnd – handle of current session;
x1, *y1*, *x2*, *y2* – diagonal coordinates;
layer – LAYER, the value of the layer must be in the range of 0 to 255;
boxtype – `BOXTYPE, t`he value of the boxtype must be in the range of 0 to 255;
elflags - [ELFLAGS], bit 15 (the rightmost bit)specifies Template data. Bit 14 specifies External data (also referred to as Exterior data). All other bits are currently unused and must be cleared to 0. If this record is omitted, all bits are assumed to be 0. The following shows an ELFLAGS record. For additional information on Template data, consult the GDSII Reference Manual.
Predefined constants:

```
//ELFLAGS
const   __int16 ELFLAGS_0=0;
const   __int16 ELFLAGS_TEMPLATE=1;
const   __int16 ELFLAGS_EXTERIOR=2;
```

plex - [PLEX], an unique positive number which is common to all elements of the plex to which this element belongs. The head of the plex is flagged by setting the seventh bit; therefore, plex numbers should be small enough to occupy only the right-most 24 bits. Predefined constant:

```
const  __int32 PLEX_HEAD=0x1000000
```

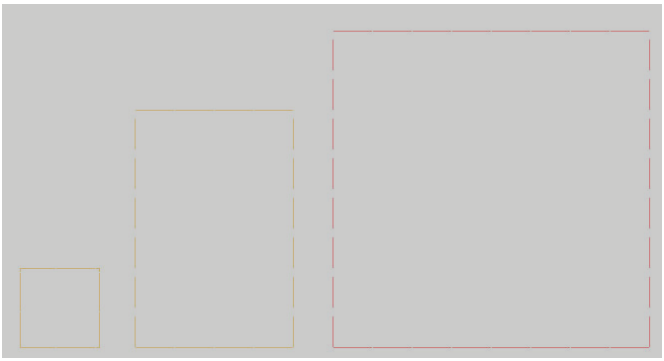
GDS_BoxWH is an extension of the GDSII stream format and builds a box. GDS_BoxWH uses BOX record.
x, *y* – coordinates of the init point (the left-bottom point);
width, *height* – width and height of the box;

GDS_Square is an extension of the GDSII stream format and builds a square. GDS_Square uses BOX record.
x, *y* – coordinates of the init point (the left-bottom point);
side – length of the square side;

Example:

```
#include "GDSIIInterface.h"
#include "GDSIIConst.h"

#pragma comment(lib, "GDSIIExporter.lib")

int main(int argc, char* argv[])
{
        HANDLE hnd;
        char* f="test.gds";

        GDS_BeginLib(f, 1e-3, 1e-9, &hnd);
          GDS_BeginStructure(hnd, "str1");
                GDS_Box(hnd, 105, 105,  205, 205,0,10, 0, 0);
                GDS_BoxWH(hnd, 250, 105,  200, 300,0,10, 0, 0);
                GDS_Square(hnd, 500, 105, 400,0,0,0,0);
          GDS_EndStructure(hnd);
        GDS_EndLib(hnd);

        return 0;
}
```

Result:



```
void GDS_Text( HANDLE hnd, int pointX, int pointY,
                const CHAR *msg, short layer,
                 short txtType, short presentation,
                short pathtype, int width,
                short strans, double mag, double angle, short
                elflags,     int plex);
```

The function creates TEXT element. In Bachus Naur Form the element can be represented as:

```
TEXT [ELFLAGS] [PLEX] LAYER TEXTTYPE
[PRESENTATION] [PATHTYPE] [WIDTH] [STRANS
[MAG] [ANGLE]] XY STRING
```

Messages can be included in a circuit with the TEXT record. The optional ELFLAGS and PLEX follow with the mandatory LAYER record after that. A TEXTTYPE record must then appear. An optional PRESENTATION record specifies the font in bits 11 and 12, the vertical presentation in bits 13 and 14 (0 for top, 1 for middle, 2 for bottom), and the horizontal presentation in bits 15 and 16 (0 for left, 1 for center, 2 for right). Optional PATHTYPE, WIDTH, STRANS, MAG, and ANGLE records may appear to affect the text. The last two records are required: an XY with a single coordinate to locate the text and a STRING record to specify the actual text.

The arguments:
hnd – handle of current session;
pointX, *pointY* – init coordinates of the text to print;
msg – text to be printed;
layer – LAYER, the value of the layer must be in the range of 0 to 255;
txtType- the value of the texttype must be in the range 0 to 255;
presentation - bits 10 and 11, taken together as a binary number, specify the font (00 means font 0, 01 means font 1, 10 means font 2, and 11 means font 3). Bits 12 and 13 specify the vertical justification (00 means top, 01 means middle, and 10 means bottom). Bits 14

and 15 specify the horizontal justification (00 means left, 01 means center, and 10 means right). Bits 0 through 9 are reserved for future use and must be cleared. If this record is omitted, then top-left justification and font 0 are assumed. The following shows a PRESENTATION record.

```
//Presentation bits
//Horizontal justification
const   __int16 PRESENT_HLEFT=0x0;
const   __int16 PRESENT_HCENTER=0x1;
const   __int16 PRESENT_HRIGHT=0x2;

//Vertical justification
const   __int16 PRESENT_VTOP=0x0;
const   __int16 PRESENT_VMIDDLE=0x4;
const   __int16 PRESENT_VBOTTOM=0x8;

//Font number
const   __int16 PRESENT_FONT0=0x0;
const   __int16 PRESENT_FONT1=0x10;
const   __int16 PRESENT_FONT2=0x20;
const   __int16 PRESENT_FONT3=0x30;
```

pathtype – see GDS_Path;

width - [WIDTH], specifies the width of the text lines in database units. A negative value for width means that the width is absolute, that is, the width is not affected by the magnification factor of any parent reference. If omitted, zero is assumed;

strans – [STRANS], bit 0 (the leftmost bit) specifies reflection. If bit 0 is set, the element is reflected about the X-axis before angular rotation. For an Aref, the entire array is reflected, with the individual array members rigidly attached. Bit 13 flags absolute magnification. Bit 14 flags absolute angle. Bit 15 (the rightmost bit) and all remaining bits are reserved for future use and must be cleared. If this record is omitted, the element is assumed to have no reflection, non-absolute magnification, and non-absolute angle. The following shows a STRANS record.

```
//STRANS
const   __int16 STRANS_0=0;
const   __int16 STRANS_ABSMAG=0x4;
const   __int16 STRANS_ABSANG=0x2;
const   __int16 STRANS_REFLECT=0x8000;
```

mag – [MAG], contains a double-precision real number (8 bytes), which is the magnification factor. If this record is omitted, a magnification factor of one is assumed.

angle – [ANGLE], angular rotation factor in CCW direction. If omitted, the default is 0. The angle of rotation is measured in degrees.

elflags - [ELFLAGS], bit 15 (the rightmost bit)specifies Template data. Bit 14 specifies External data (also referred to as Exterior data). All other bits are currently unused and must be cleared to 0. If this record is omitted, all bits are assumed to be 0. The following shows an ELFLAGS record. For additional information on Template data, consult the GDSII Reference Manual.

Predefined constants:

```
//ELFLAGS
const   __int16 ELFLAGS_0=0;
const   __int16 ELFLAGS_TEMPLATE=1;
const   __int16 ELFLAGS_EXTERIOR=2;
```

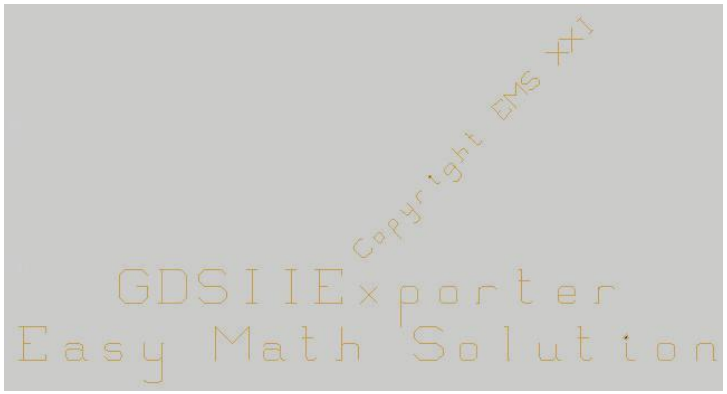
plex - [PLEX], an unique positive number which is common to all elements of the plex to which this element belongs. The head of the plex is flagged by setting the seventh bit; therefore, plex numbers should be small enough to occupy only the right-most 24 bits. Predefined constant:

```
const  __int32 PLEX_HEAD=0x1000000
```

Example:

```
#include "GDSIIInterface.h"
#include "GDSIIConst.h"

#pragma comment(lib, "GDSIIExporter.lib")

int main(int argc, char* argv[])
{
        HANDLE hnd;
        char* f="test.gds";

        GDS_BeginLib(f, 1e-3, 1e-9, &hnd);
                GDS_BeginStructure(hnd, "str1");
                        GDS_Text( hnd, 100, 100, "Easy Math Solution",
1, 4,
                        PRESENT_HCENTER |PRESENT_VBOTTOM,
PATHTYPE1, 0, STRANS_0, 0.1, 0, 0, 0);
                        GDS_Text( hnd, 100, 300, "GDSIIExporter", 1, 4,
                        PRESENT_HCENTER |PRESENT_VBOTTOM,
PATHTYPE1, 0, STRANS_0, 0.1, 0, 0, 0);
                        GDS_Text( hnd, 100, 500, "Copyright EMS XXI",
1, 4,
                        PRESENT_HLEFT |PRESENT_VBOTTOM,
PATHTYPE1, 0, STRANS_0, 0.05, 45, 0, 0);
                GDS_EndStructure(hnd);
        GDS_EndLib(hnd);

        return 0;
}
```

Result:



void GDS_Sref(HANDLE hnd, const CHAR *strName, int x, int y,
short strans, double mag, double angle,
short elflags, int plex);

The function creates a SREF structure (structure reference element). In Bachus Naur Form the element can be represented as:

```
SREF [ELFLAGS] [PLEX] SNAME [STRANS [MAG]
[ANGLE]] XY
```

Hierarchy is achieved by allowing structure references (instances) to appear in other structures. The SREF record indicates a structure reference and is followed by the optional ELFLAGS and PLEX records. The SNAME record then names the desired structure and an XY record contains a single coordinate to place this instance. It is legal to make reference to structures that have not yet been defined with STRNAME.

Prior to the XY record there may be optional transformation records. The STRANS record must appear first if structure transformations are desired. Its parameter has bit flags that indicate mirroring in *x* before rotation (if bit 1 is set), the use of absolute magnification (if bit 14 is set), and the use of absolute rotation (if bit 15 is set). The magnification and rotation amounts may be specified in the optional MAG and ANGLE records. The rotation angle is in counterclockwise degrees.

The arguments:
hnd – handle of current session;
strName – a name of structure to be referenced.
x,y – a point coordinate where the structure to be printed.
strans – [STRANS], bit 0 (the leftmost bit) specifies reflection. If bit 0 is set, the element is reflected about the X-axis before angular rotation. For an Aref, the entire array is reflected, with the individual array members rigidly attached. Bit 13 flags absolute magnification. Bit 14 flags absolute angle. Bit 15 (the rightmost bit) and all remaining bits are reserved for future use and must be cleared. If this record is omitted, the element is assumed to have no reflection, non-absolute magnification, and non-absolute angle. The following shows a STRANS record.

```
//STRANS
const   __int16 STRANS_0=0;
const   __int16 STRANS_ABSMAG=0x4;
const   __int16 STRANS_ABSANG=0x2;
const   __int16 STRANS_REFLECT=0x8000;
```

mag – [MAG], contains a double-precision real number (8 bytes), which is the magnification factor. If this record is omitted, a magnification factor of one is assumed.
angle – [ANGLE], angular rotation factor in CCW direction. If omitted, the default is 0. The angle of rotation is measured in degrees.
elflags - [ELFLAGS], bit 15 (the rightmost bit)specifies Template data. Bit 14 specifies External data (also referred to as Exterior data). All other bits are currently unused and must be cleared to 0. If this record is omitted, all bits are assumed to be 0. The following shows an ELFLAGS record. For additional information on Template data, consult the GDSII Reference Manual.
Predefined constants:

```
//ELFLAGS
const   __int16 ELFLAGS_0=0;
const   __int16 ELFLAGS_TEMPLATE=1;
const   __int16 ELFLAGS_EXTERIOR=2;
```

plex - [PLEX], a unique positive number which is common to all elements of the plex to which this element belongs. The head of the plex is flagged by setting the seventh bit; therefore, plex numbers should be small enough to occupy only the right-most 24 bits. Predefined constant:

```
const   __int32 PLEX_HEAD=0x1000000
```

Example:

```
#include "GDSIIInterface.h"
#include "GDSIIConst.h"

#pragma comment(lib, "GDSIIExporter.lib")

int main(int argc, char* argv[])
{
        HANDLE hnd;
        char* f="test.gds";

        GDS_BeginLib(f, 1e-3, 1e-9, &hnd);
                GDS_BeginStructure(hnd,"str3");
                        GDS_Sref(hnd, "str1", 1000,
                1000,0,1,30,ELFLAGS_TEMPLATE,0);
                GDS_EndStructure(hnd);

                GDS_BeginStructure(hnd, "str1");
                        GDS_Text( hnd, 100, 100, "Easy Math Solution",
1, 4,
                        PRESENT_HCENTER |PRESENT_VBOTTOM,
PATHTYPE1, 0, STRANS_0, 0.1, 0, 0, 0);
                        GDS_Text( hnd, 100, 300, "GDSIIExporter", 1, 4,
                        PRESENT_HCENTER |PRESENT_VBOTTOM,
PATHTYPE1, 0, STRANS_0, 0.1, 0, 0, 0);
                        GDS_Text( hnd, 100, 500, "Copyright EMS XXI",
1, 4,
                        PRESENT_HLEFT |PRESENT_VBOTTOM,
PATHTYPE1, 0, STRANS_0, 0.05, 45, 0, 0);
                GDS_EndStructure(hnd);
        GDS_EndLib(hnd);

        return 0;
}
```

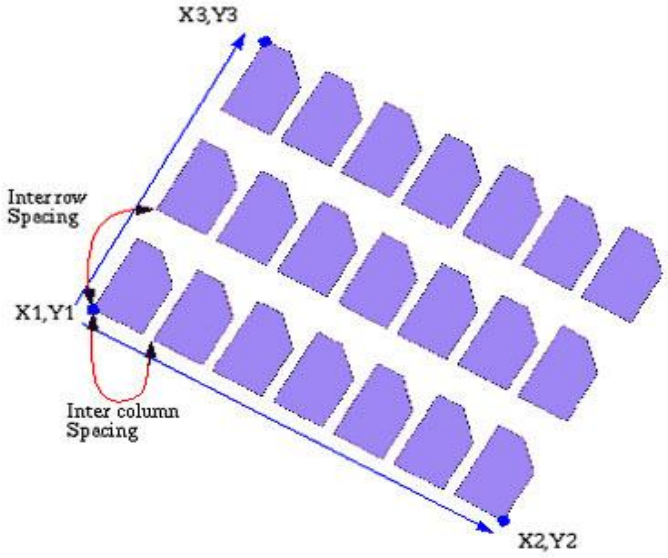
void GDS_Aref(HANDLE hnd, const char *strName, int *x, int *y, short col, short row, short strans, double mag, double angle, short elflags, int plex);

The function creates an AREF structure (structure array reference element). In Bachus Naur Form the element can be represented as:

```
AREF [ELFLAGS] [PLEX] SNAME  [STRANS [MAG]
[ANGLE]] COLROW XY
```

For convenience, an array of structure instances can be specified with the AREF record. Following the optional ELFLAGS and PLEX records comes the SNAME to identify the structure being arrayed. Next, the optional transformation records STRANS, MAG, and ANGLE give the orientation of the instances. A COLROW record must follow to specify the number of columns and the number of rows in the array. The final record is an XY with three points: the coordinate of the corner instance, the coordinate of the last instance in the columnar direction, and the coordinate of the last instance in the row direction. From this information, the amount of instance overlap or separation can be determined. Note that flipping arrays (in which alternating rows or columns are mirrored to abut along the same side) can be implemented with multiple arrays that are interlaced and spaced apart to describe alternating rows or columns.

The arguments:
hnd – handle of current session;
strName – a name of the structure to be referenced.
x,y – an array of the coordinates where the structure array to be printed. An Aref has exactly three coordinates. In an Aref, the first coordinate is the array reference point (origin point). The other two coordinates are already rotated, reflected as specified in the STRANS record (if specified). So in order to calculate the inter column and inter row spacing, the coordinates must be mapped back to their original position, or the vector length (x1,y1-> x3,y3) must be divided by the number of row etc. . The second coordinate locates a position, which is displaced from the reference point by the inter-column spacing times the number of columns. The third coordinate locates a position, which is displaced from the reference point by the inter-row spacing times the number of rows. For an example of an array lattice see the next picture.



col, row – columns and rows for an AREF. Two 2 byte integers. The first is the number of columns. The second is the number of rows. Neither may exceed 32767.

strans – [STRANS], bit 0 (the leftmost bit) specifies reflection. If bit 0 is set, the element is reflected about the X-axis before angular rotation. For an Aref, the entire array is reflected, with the individual array members rigidly attached. Bit 13 flags absolute magnification. Bit 14 flags absolute angle. Bit 15 (the rightmost bit) and all remaining bits are reserved for future use and must be cleared. If this record is omitted, the element is assumed to have no reflection, non-absolute magnification, and non-absolute angle. The following shows a STRANS record.

```
//STRANS
const   __int16 STRANS_0=0;
const   __int16 STRANS_ABSMAG=0x4;
const   __int16 STRANS_ABSANG=0x2;
const   __int16 STRANS_REFLECT=0x8000;
```

mag – [MAG], contains a double-precision real number (8 bytes), which is the magnification factor. If this record is omitted, a magnification factor of one is assumed.

angle – [ANGLE], angular rotation factor in CCW direction. If omitted, the default is 0. The angle of rotation is measured in degrees.

elflags - [ELFLAGS], bit 15 (the rightmost bit)specifies Template data. Bit 14 specifies External data (also referred to as Exterior data). All other bits are currently unused and must be cleared to 0. If this record is omitted, all bits are assumed to be 0. The following shows an ELFLAGS record. For additional information on Template data, consult the GDSII Reference Manual.

Predefined constants:

```
//ELFLAGS
const   __int16 ELFLAGS_0=0;
const   __int16 ELFLAGS_TEMPLATE=1;
const   __int16 ELFLAGS_EXTERIOR=2;
```

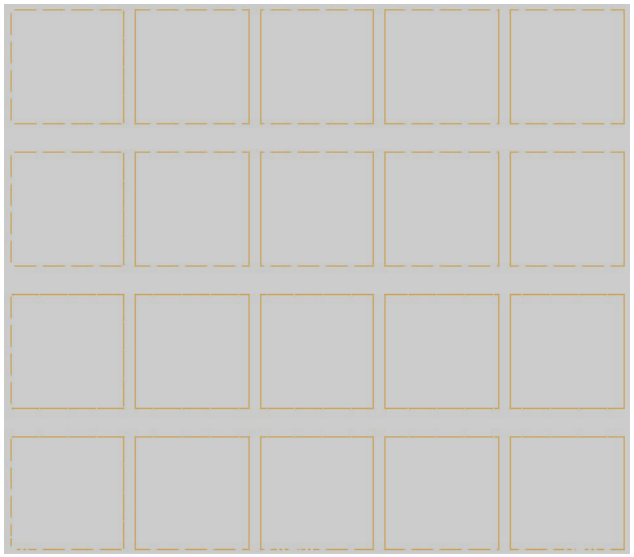
plex - [PLEX], an unique positive number which is common to all elements of the plex to which this element belongs. The head of the plex is flagged by setting the seventh bit; therefore, plex numbers should be small enough to occupy only the right-most 24 bits. Predefined constant:

```
const  __int32 PLEX_HEAD=0x1000000
```

Example:

```
#include "GDSIIInterface.h"
#include "GDSIIConst.h"

#pragma comment(lib, "GDSIIExporter.lib")

int main(int argc, char* argv[])
{
        HANDLE hnd;
        char* f="test.gds";
        int xar[]={0,1100,0};
        int yar[]={0,0,1000};


        GDS_BeginLib(f, 1e-3, 1e-9, &hnd);
                GDS_BeginStructure(hnd, "str2");
                        GDS_Aref(hnd, "ur11",  xar, yar,5,4,0,1,0,0,0);
                GDS_EndStructure(hnd);

                GDS_BeginStructure(hnd, "ur11");
                        GDS_Box(hnd, 0,0,200,200, 0,10, 0, 0);
                GDS_EndStructure(hnd);

        GDS_EndLib(hnd);

        return 0;
}
```

Result:



void GDS_Node(HANDLE hnd, int *pointsX, int *pointsY, short pointsNum, short layer, short nodeType, short elflags, int plex);

The function creates a NODE structure. In Bachus Naur Form the element can be represented as:

```
NODE [ELFLAGS] [PLEX] LAYER NODETYPE XY
```

Electrical nets may be specified with the NODE record. The optional ELFLAGS and PLEX records follow and the required LAYER record is next. A NODETYPE record must appear with a zero argument, followed by an XY record with one to 50 points that identify coordinates on the electrical net. The information in this element is not graphical and does not

affect the manufactured circuit. Rather, it is for other CAD systems that use topological information.

The arguments:
hnd – handle of current session;
pointsX, *pointsY* – arrays of the XY;
pointsNum – number of points in the arrays *pointsX* and *pointsY*. Node elements may have a minimum of 1 and a maximum of 50 coordinates;
layer – LAYER, the value of the layer must be in the range of 0 to 255;
nodeType - the value of the nodetype must be in the range of 0 to 255;
elflags - [ELFLAGS], bit 15 (the rightmost bit)specifies Template data. Bit 14 specifies External data (also referred to as Exterior data). All other bits are currently unused and must be cleared to 0. If this record is omitted, all bits are assumed to be 0. The following shows an ELFLAGS record. For additional information on Template data, consult the GDSII Reference Manual.
Predefined constants:

```
//ELFLAGS
const   __int16 ELFLAGS_0=0;
const   __int16 ELFLAGS_TEMPLATE=1;
const   __int16 ELFLAGS_EXTERIOR=2;
```

plex - [PLEX], a unique positive number which is common to all elements of the plex to which this element belongs. The head of the plex is flagged by setting the seventh bit; therefore, plex numbers should be small enough to occupy only the right-most 24 bits. Predefined constant:

```
const  __int32 PLEX_HEAD=0x1000000
```

Example:

```
#include "GDSIIInterface.h"
#include "GDSIIConst.h"

#pragma comment(lib, "GDSIIExporter.lib")

int main(int argc, char* argv[])
{
        HANDLE hnd;
        char* f="test.gds";
        int xar[]={0,1100,0};
        int yar[]={0,0,1000};


        GDS_BeginLib(f, 1e-3, 1e-9, &hnd);
                GDS_BeginStructure(hnd, "str2");
                        GDS_Node(hnd, xar, yar,4, 0,0,0,0);
                GDS_EndStructure(hnd);
        GDS_EndLib(hnd);

        return 0;
}
```

Error Handling. Examples.

Each function from the interface returns integer value as indicator of SUCCESS or FAILURE. If no error happens a function returns 0 (zero). If result is not equal 0, then error happened and user can get the error message using:

```
void GDS_GetError(int errCode, char* errBuffer, int buferSize);
```

The arguments:
errCode- error code to get message;
errBuffer – pointer to buffer where message will be copied;
buferSize – size of allocated buffer for error message;

Example:

```
#include "GDSIIInterface.h"
#include "GDSIIConst.h"

#pragma comment(lib, "GDSIIExporter.lib")

int main(int argc, char* argv[])
{
        HANDLE hnd;
        char* f="test.gds";
        int errCode = eErrNoError;

        int x[]={110,210,327,200,600,700,880,1000};
        int y[]={0,100,250,150,300,440,900,800};

        if(eErrNoError != (errCode = GDS_BeginLib(f, 1e-3, 1e-9, &hnd)))
                goto error;

         if(eErrNoError != (errCode = GDS_BeginStructure(hnd, "str1")))
                goto error;
                if(eErrNoError != (errCode = GDS_Path(hnd, x, y, 8 ,10,
1,63, PATHTYPE1, 0, 0)))
                        goto error;
         if(eErrNoError != (errCode = GDS_EndStructure(hnd)))
                goto error;

        if(eErrNoError != (errCode = GDS_EndLib(hnd)))
                goto error;


        return 0;
error:
        const int buffSize =200;
        char errMessage[buffSize];

        GDS_GetError(errCode, errMessage, buffSize);
}
```

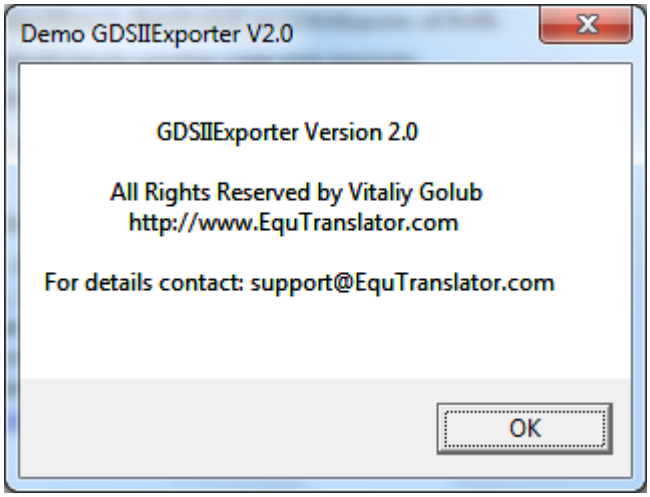
Full list of error messages can be found in GDSIIConst.h header file:

```
enum ERR_CODE {
eErrNoError = 0,                        //"No error."
eErrHandle,                     //"The handle is not defined."
eErrDoubleOverRange,            //"Error: Source double value is over range."
eErrOpenGDSIIFile,              //"Error: Cannot open GDSII file."
eErrStartAnotherProc,           //"Error: Cannot start another export process."
ErrProcAlreadyStop,             //"Error: Process is already stopped."
eErrStructNotClosed,            //"Error: Structure is not closed."
eErrStructAlreadyStarted,       //"Error: Structure is already opened."
eErrStructNotOpened,            //"Error: Structure is not opened."
eErrStructAlreadyInserted,      //"Error: Structure with this name is already
inserted."
eErrMuchBoundaryPoints,         //"Error: Too many points for BOUNDARY
structure. "
eErrFewBoundaryPoints,          //"Error: Not enough points for BOUNDARY
structure. "
eErrFewPathPoints,              //"Error: Not enough points for PATH structure. "
eErrStringIsNULL,               //"Error: The string is not defined. "
eErrStringSizeIsOver,           //"Error: The string size is more than 512
characters. "
eErrTextTypeLimit,              //"Error: TEXTTYPE value should be in the range
0 to 255. "
eErrMaxNodePoints,              //"Error: Too many points for NODE record. "
```

```
eErrMinNodePoints,          //"Error: Not enough points for NODE record. "
eEndElement} ;
```

Demo of GDSIIExporter

Demo for GDSIIExporter is distributed as GDSIIExporter.dll both x86 and x64. It's fully functionally binary with dialog window with copyright information:



Have buying the product you get binary without prompt window.