INTERNATIONAL STANDARD

ISO/IEC 8651-4

Second edition
1995-06-01

Information technology — Computer graphics — Graphical Kernel System (GKS) language bindings —

Part 4: C

Technologies de l'information — Infographie — Interfaces langage avec GKS —

Partie 4: C

Reference number
ISO/IEC 8651-4:1995(E)

Contents

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 8651-4 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology,* Subcommittee SC 24, *Computer graphics and image processing*.

This second edition cancels and replaces the first edition (ISO/IEC 8651-4:1991), which has been technically revised.

ISO/IEC 8651-4 consists of the following parts, under the general title *Information technology — Computer graphics — Graphical Kernel System (GKS) language bindings*:

- *Part 1: FORTRAN*
- *Part 2: Pascal*
- *Part 3: Ada*
- *Part 4: C*

Annexes A to F of this part of ISO/IEC 8651 are for information only.

Introduction

The Graphical Kernel System (GKS) functional description is registered as ISO/IEC 7942-1:1994. As explained in the Scope and Field of Application of ISO/IEC 7942-1, that International Standard is specified in a language independent manner and needs to be embedded in language dependent layers (language bindings) for use with particular programming languages.

The purpose of this part of ISO/IEC 8651 is to define a standard binding for the C computer programming language.

Information technology — Computer graphics — Graphical Kernel System (GKS) language bindings —

Part 4:
C

1 Scope

The Graphical Kernel System (GKS), ISO/IEC 7942–1:1994 , specifies a language independent nucleus of a graphics system. For integration into a programming language, GKS is embedded in a language dependent layer obeying the particular conventions of that language. This part of ISO/IEC 8651 specifies such a language dependent layer for the C language.

2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 8651. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO/IEC 8651 are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 7942–1:1994, *Information technology – Computer graphics and image processing – Graphical Kernel System (GKS) – Part 1 : Functional description.*
ISO/IEC 9899:1990, *Programming languages – C.*
ISO/IEC TR 9973:1994, *Information technology – Computer graphics and image processing – Procedures for registration of graphical Items.*

3 The C language binding

The C language binding of GKS shall be as described in clauses 3 to 7.

3.1 Classification and designation

This part of ISO/IEC 8651 incorporates the rules of conformance defined in the GKS Standard (ISO/IEC 7942-1) for GKS implementations, with those additional requirements specifically defined for C bindings in GKS.

The following criteria shall determine conformance of an implementation to this part of ISO/IEC 8651:

In order to conform, an implementation of the C binding of GKS shall implement GKS as specified in ISO/IEC 7942–1. It shall make visible all of the declarations in the C binding specified in this part of ISO/IEC 8651 for a specific level of C.

Thus, for example, the syntax of the function names shall be precisely as specified in the binding and parameters shall be of the data types stated in the binding.

3.2 Functions versus macros

An implementation may substitute macros for functions. However, the macros shall be designed so that side-effects work properly. In general, a macro cannot be used to replace the error handling function **`gerr_hand`**. See also 3.10.

3.3 Character strings

The C language represents character strings as an array of characters terminated by the null character (i.e. `'\0'`). This means that the null character is not usable as a printable character.

3.4 Function identifiers

The function names of GKS are all mapped to C functions which begin with the letter **`g`**. Words and phrases used in the GKS function names are often abbreviated in the representation and are always separated with the underscore character **`"_"`**. The set of such abbreviations is given in 4.2, and the resulting C function names are listed in 4.3. For example, the abbreviation for the GKS function DELETE SEGMENT FROM WORKSTATION is **`gdel_seg_ws`**. **`del`**, **`seg`**, and **`ws`** are abbreviations for DELETE, SEGMENT, and WORKSTATION. The conjunctive FROM is mapped to the null string.

The C language (ISO/IEC 9899) requires that compilers recognize internal identifiers which are distinct in at least 31 characters. That standard also requires that external identifiers (i.e. those seen by the linker) be recognized to a minimum of six characters, independent of case.

Implementations which run in environments where two distinct C internal identifiers would be equivalent, if they were both external identifiers, shall include a set of object-like macros in the header which equate the long names to a set of short names. A possible set of short names for a compiler that accepts only 8 characters for external definitions may be found in annex C.

3.5 Registration

ISO/IEC 7942 reserves certain value ranges for registration[1] as graphical items. The registered graphical items will be bound to the C programming language (and other programming languages). The registered item binding will be consistent with the binding presented in this part of ISO/IEC 8651.

[1] For the purpose of this part of ISO/IEC 8651 and according to the rules for the designation and operation of registration authorities in the ISO/IEC Directives, the ISO/IEC council has designated the National Institute of Standards and Technology (Institute of Computer Sciences and Technology), A-266 Technology Building, Gaithersburg, MD 20899, USA to act as registration authority.

3.6 Identifiers for graphical items

Generalized Drawing Primitives and Escape functions are referenced via identifiers. This part of ISO/IEC 8651 specifies the format of the identifiers but it does not specify the registration of the identifiers. The identifiers are used as arguments to the functions **ggdp** and **gescape**.

An implementation may also represent GDPs and Escapes as separate functions, but this is not required.

There are two formats for these identifiers. One format is for registered GDPs and Escapes and the other format is for unregistered GDPs and Escapes.

The format for registered GDP identifiers is:

```
#define     GGDP_Rn     (n)         /* where 'n' is the registered GDP
                                       identifier */
```

The format for unregistered GDP identifiers is:

```
#define     GGDP_Un     (-n)        /* where 'n' is implementation
                                       dependent */
```

The format for registered Escape function identifiers is:

```
#define     GESCAPE_Rn  (n)         /* where 'n' is the registered
                                       Escape identifier */
```

The format for unregistered Escape function identifiers is:

```
#define     GESCAPE_Un  (-n)        /* where 'n' is implementation
                                       dependent */
```


3.7 Return values

All GKS/C functions have return value **void**.

3.8 Headers

3.8.1 gks.h

C provides a mechanism to access information stored in a header via the **#include** preprocessing directive. Clause 5 of this part of ISO/IEC 8651 describes the data types that shall be defined in the header **gks.h** which shall be included in any application program that intends to use GKS via the C binding.

This part of ISO/IEC 8651 uses the data type **size_t** (inter alia as a field in the data type **Gdata**). The type **size_t** is environment-defined (i.e. **unsigned long** or **unsigned int**) and is defined in the headers **<stdio.h>, <stddef.h>, <stdlib.h>, <string.h>, <time.h>**.

Additional implementation-dependent items may be placed in this header if needed. These items should start with the sentinel "G" or "g", as far as applicable.

The header **gks.h** shall also contain external declarations for all GKS/C functions because they have a **void** return type. For example, the declaration for the function **gopen_gks** would look like this:

```
extern   void   gopen_gks(const char *err_file, size_t memory);
```

3.8.2 gks_compat.h

For application programs which used to run on top of the 1991 edition of this part of ISO/IEC, the header **gks_compat.h** is provided. **gks_compat.h** includes GKS/C data types that are no longer supported, as well as external declarations for all GKS/C functions that are no longer supported. Implementations of this part of ISO/IEC 8651 shall support these functions in a compatibility layer, according to the guidelines in Annex G of ISO/IEC 7942–1:1994.

3.9 Memory management

The application shall allocate the memory needed for the data returned by the implementation. In general, the application will allocate a C structure and pass a pointer to that structure to an inquiry routine, which will then place information into the structure. However, a number of inquiry functions return variable length data, the length of which is not known *a priori* by the application.

These functions fall into two classes. One class of functions returns a simple, homogeneous, list of items. For example, the function INQUIRE LIST OF MARKER INDICES returns a list of the available marker indices. The other class returns complex, heterogeneous data structures. For example, the function INQUIRE GKS STATE LIST ENTRY returns a piece of the GKS state which may include several data structures of different length. The binding of these two classes of functions is described in detail below. Subclause 3.10 describes the errors that can be invoked during execution of functions which use the memory management policy.

3.9.1 Functions which return simple lists

Inquiry functions which return a list of items are bound such that the application can inquire about a portion of the list. This list is a subset of the implementation's internal list and is called the application's list. This allows the application to process the implementation's list in a piecewise manner rather than all at once.

The application allocates the memory for a list and passes that list to the implementation. The implementation places the results of the inquiry into the list. In order to support this policy of memory management, three additional parameters have been added to functions which return lists:

a) **num_elems_appl_list**: An integer input parameter which is the length of the application's list. The value of **num_elems_appl_list** indicates the number of items (i.e. list elements) which will fit into the application list. A value of 0 is valid and allows the application to determine the size of the implementation's list (which is returned via **num_elems_impl_list**) without having the implementation return any of the elements of its list. If **num_elems_appl_list** is negative, **GE_APPL_LIST_LENGTH_LT_ZERO** is returned as the value of the error indicator parameter.

b) **start_ind**: An integer input parameter which is an index into the implementation's list. (Index 0 is the first element of both the implementation's and application's list.) **start_ind** indicates the first item in the implementation's list that is copied into index 0 of the application's list. Items are copied sequentially from the implementation's list into the application's list until the application's list is full or there are no more items in the implementation's list. If **start_ind** is out of range, error **GE_START_IND_INVAL** is returned as the value of the error indicator parameter.

c) **num_elems_impl_list**: An output parameter which is a pointer to an integer. The implementation stores into this parameter the number of items that are in the implementation's list.

In annex D, a possible underlying mechanism is described.

3.9.2 Functions which return complex data structures

The data returned by inter alia the ESCAPE function, the AWAIT INPUT function and the functions which return state lists or description tables can be complex in structure. They cannot be represented by a simple list of items. It would be an onerous task for the application to have to allocate and prepare data structures for these routines. In order to facilitate this task of using these inquiry functions, the binding defines a new resource, called a *Store*, to manage the memory for these functions.

The *Store* resource is opaque to the application.The application does not know the structure of the *Store* or how it is implemented. The *Store* is defined as a **void *.** This part of ISO/IEC 8651 defines two new functions which create (in CREATE STORE, bound as **gcreate_store**) and delete (in DELETE STORE, bound as **del_store**) a *Store*.

A *Store* is used by the implementation to manage the memory needed by the functions which return complex data structures. Without specifying an implementation of a *Store*, it is safe to say that it will contain

and control memory needed to hold the data returned by these functions and also contain some bookkeeping information about the contents and size of the memory.

The semantics of the *Store* resource provide two levels of memory management. The implementation is responsible for managing the memory at a low level because it uses, reuses, allocates and deallocates memory from the system in order to return information to the application. But the application is ultimately responsible for managing the memory at a high level because it creates and deletes *Stores*.

A *Store* is passed as a parameter to a function returning complex data structures. Another parameter to this function is a pointer to a pointer to a structure which defines the format of the returned data. The *Store* contains memory for the structure and any additional memory referenced by fields within the structure. The application accesses the returned data through its pointer to the structure. It does not use the *Store* to access the data.

A *Store* continues to hold the information from the function until the *Store* is deleted by the DELETE STORE function or until the *Store* is used as an argument to a subsequent function, which returns complex data structures. At that time, the old information is replaced with the new. Thus multiple calls to functions overwrite the contents of a *Store*. A *Store* only contains the results of the last function.

This part of ISO/IEC 8651 defines two errors that can occur when using or creating a *Store*; these errors are described in 6.2. For most functions using a *Store*, these and other errors are returned via the "error indicator" parameter. However, the function ESCAPE does not have an error indicator parameter. For this function, the error reporting mechanism is used when an error is encountered. For this function, the implementation shall, in addition to reporting the error, set the pointer to the returned data to NULL when an error occurs. See the binding of these functions for more information.

The definitions for the functions CREATE STORE and DELETE STORE follow:

CREATE STORE

Parameters:

Out	error indicator	I
Out	store	STORE

Effect: Creates a *Store* and returns a handle to it in the output parameter *store*. If the *Store* cannot be created, the error indicator is set to one of the following error values:

GE_GKS_NOT_OPEN	GKS not in proper state; GKS shall be in the state (GKOP, *)
GE_ERR_ALLOC_STORE	Error while allocating Store

Errors: None.

DELETE STORE

Parameters:

Out	error indicator	I
Out	store	STORE

Effect: Deletes the *Store* and all internal resources associated with it. If there is not an error, the parameter *store* will be set to NULL to signify that it is no longer valid. If an error is detected, the error indiocator is set to one of the following values:

GE_GKS_NOT_OPEN	GKS not in proper state; GKS shall be in the state (GKOP, *)

Errors: None.

In 7.2.13.2, the C specification of these functions is given. In annex D, a possible underlying mechanism is

illustrated.

3.10 Error handling

3.10.1 Application supplied error handlers

User-defined error handlers shall accept the same arguments as the standard error handler. The user-defined error handler is specified by the utility function (see also 7.2.13.2)

SET ERROR HANDLER

Parameters:

In	New error handling function	Function
Out	Old error handling function	Function

Effect: Sets the GKS error handling function to *New error handling function* and returns the current error handling function to *Old error handling function.*

Errors: None.

Application defined error handling functions accept the same arguments as the standard error handler. They may invoke the standard error logging function ERROR LOGGING.

ISO/IEC 7942 defines the initial error handling function to be ERROR HANDLING, that is, the value of the parameter *Old error handling function* points to ERROR HANDLING, when SET ERROR HANDLER is called for the first time.

When the application changes the error handling function, the implementation will invoke the new function when an error is detected. If the application calls the default error handling function ERROR HANDLING, ERROR HANDLING will always call the function ERROR LOGGING.

If *New error handler* is not a valid pointer, the error handling will automatically be done by the standard error handler ERROR HANDLING.

User-defined error handlers may invoke the standard error logging function ERROR LOGGING.

3.10.2 Error codes

Hard coding numbers into a program decreases its maintainability. Therefore, this part of ISO/IEC 8651 defines a set of constants for the GKS error numbers. Each error constant begins with the characters **`GE_`**. See also 6.2 for the error macros.

3.10.3 C-specific GKS errors

This part of ISO/IEC 8651 defines some additional error messages. In 6.2 the numbers and their macros are given.

3.11 Colour representations and specifications

GKS defines 4 colour models (RGB, CIE L*u*v* 1976, HLS and HSV) of which RGB and CIE L*u*v* are mandatory. For each of these models, a colour specification is defined (**`Grgb, Gcieluv, Ghsv, Ghls`**). The colour representation and specification are defined in 5.4 by the types **`Gcolr_rep`** and **`Gcolr_specif.`**

3.12 Colour characteristics

GKS defines the colour characteristics as a basic type. The colour characteristics is defined in this part of ISO/IEC 8651 by the data type **`Gcolr_chars`**, which is documented in 5.3.

3.13 Storage of multi-dimensional arrays

3.13.1 Storage of 2*3 matrices

The entries of **Gtran_matrix** data types shall be stored such that the segment transformation is defined by

```
Tp.x  =  mat[0,0]*p.x  +  mat[0,1]*p.y  +  mat[0,2];
Tp.y  =  mat[1,0]*p.x  +  mat[1,1]*p.y  +  mat[1,2];
```

where **p** is a 2D point, **Tp** its transformation and **mat** is of type **Gtran_matrix**.

3.13.2 Storage of conics in 3*3 matrices

The entries of **Gconic_matrix** data types shall be stored such that the conic is defined by

$$a[0,0]x^2 + (a[0,1] + a[1,0])\ xy + a[1,1]y^2$$
$$+ (a[0,2] + a[2,0])x + (a[1,2] + a[2,1])y + a[2,2] = 0$$

where **a** is of type **Gconic_matrix**.

3.13.3 Storage of colour arrays

The entries of **Gpat_rep** data types shall be stored such that the colour specifier at the (i,j)-th entry is given by

$xxx_{i,j}$ = **colr_rect.dir_colr.xxx[i + DX*j];** i = 0, . . . , DX - 1; j

$colr\_ind_{i,j}$ = **colr_rect.colr_array[i + DX*j];**

DX = **colr_rect.dims.size_x;**

DY = **colr_rect.dims.size_y;**

where xxx = (rgb|cieluv|hls|hsv) is of type **Gxxx** and **colr_rect** is of type **Gpat_rep**.

3.14 Compatibility with the 1991 edition

In the second (1994) edition of ISO/IEC 7942 some functions and data types of the 1985 edition have been overtaken or deprecated. They are documented in the informative annex G. Examples are the GKSM functions, which have been overtaken by AUDIT functions.

In this edition of ISO/IEC 8651–4 *all* functions and *all* data types of of the 1991 edition (the C binding of the 1985 edition of ISO/IEC 7942) have been preserved. The support of overtaken or deprecated functions or data types falls into two categories:

1) *Functions which have been renamed (e.g. INQUIRE TEXT EXTENT → GET TEXT EXTENT) or which have been incorporated into more compact functions (e.g. the OUTPUT functions).*
   These functions and their related data types are documented in clauses 5 and 7. For example, the function CREATE OUTPUT PRIMITIVE has been bound to the C function **gcreate_out_prim**. Besides that, however, the output functions **gpolyline, gpolyline_set, gnurb_set, gconic_sec_set, gpolymarker, gfill_area, gfill_area_set, gell_sec_set, gell_seg_set, gell_disc_set, gclosed_nurb_set, gtext, gcell_array, gdesign, ggdp** have been defined.

2) *Functions which have been deprecated (INQUIRE GKS LEVEL) or which have been overtaken by other functions (GKSM functions, for example) .*
   These functions are documented in annex E. They will be deleted at the next edition of this part of ISO/IEC 8651.

4 Tables

4.1 Abbreviation policy in construction of identifiers

In the construction of the several data types, function names, etc., the following policy is applied:

1) All identifiers in the C binding are abbreviated using the same abbreviations for every component and using underscores to denote blanks.
2) The plural of an expression is constructed by adding an "s" after its abbreviation; so, for example, "vector" is abbreviated to "vec" and "vectors" is abbreviated to "vecs"; if an expression is mapped to NULL, so will be its plural.
3) Digits are also preceded by underscores.
4) The word REALIZED is not abbreviated in the second field of the enumeration data type **`Ginq_type`**; in all other cases it is abbreviated using the list in 4.2.
5) Construction of GKS/C identifiers:
   a) Function names:
   "**`g`**" (lower case) followed by abbreviated function name in lower case;
   b) Data types:
   "**`G`**" (upper case) followed by abbreviated data type in lower case;
   c) Fields of data types; the following refinements are used: "redundant" (words in the field name that are identical to those in the structure name) parts are omitted, if the context allows this; thus the linewidth in the field of **`Gline_bundle`** is abbreviated to **`width`**, because the context makes clear which width is used;
   d) Function macros:
   "**`Gfn_`**" followed by abbreviation of function name;
   e) Error macros:
   "**`GE_`**" followed by some abbreviated expression;
   f) Fields of enumeration types:
   "**`G`**" (upper case) followed by a prefix followed by an abbreviation of the field name; this prefix is constant for each enumeration field; all the fields are in upper case.

4.2 Table of abbreviations used

In this table, only words which are abbreviated are listed. They are used for

function names;
data types;
fields of data types;
error macros.

The word "NULL" denotes those words which are deleted completely when forming function names or data types.

Word or Phrase	Abbreviation
CIE L*u*v* (colour model)	cieluv
accumulate	accum
actual	act
addition	add
alignment	align
allocate	alloc
and	NULL