

## **Libraries**

COLLABORATORS

	TITLE : Libraries		
ACTION	NAME	DATE	SIGNATURE
WRITTEN BY		July 23, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>1</b>	<b>Libraries</b>	<b>1</b>
1.1	Amiga® RKM Libraries: 34 Keymap Library . . . . .	1
1.2	34 Keymap Library / Keymap Functions . . . . .	2
1.3	34 / Keymap Functions / Asking For the Default Keymap . . . . .	3
1.4	34 / Keymap Functions / Setting the Default Keymap . . . . .	3
1.5	34 / Keymap Functions / Accessing the Keymap For the Current Console . . . . .	3
1.6	34 / Keymap Functions / Mapping Key Codes To ANSI Strings . . . . .	4
1.7	34 / Keymap Functions / Mapping ANSI Strings To Key Codes . . . . .	5
1.8	34 / Keymap Functions / Details Of the Keymap Structure . . . . .	5
1.9	34 // Details Of the Keymap Structure / LoKeyMap and HighKeyMap . . . . .	6
1.10	34 // Details Of Keymap Structure / LoKeyMapTypes and HiKeyMapTypes . . . . .	7
1.11	34 // Details Of the Keymap Structure / More About Qualifiers . . . . .	7
1.12	34 // Details Of the Keymap Structure / String Output Keys . . . . .	8
1.13	34 // Details Of the Keymap Structure / Capsable Bit Tables . . . . .	10
1.14	34 // Details Of the Keymap Structure / Repeatable Bit Tables . . . . .	10
1.15	34 / Keymap Functions / Key Map Standards . . . . .	10
1.16	34 / Keymap Functions / Dead-Class Keys . . . . .	11
1.17	34 / Keymap Functions / Double-Dead Keys . . . . .	14
1.18	34 Keymap Library / Keyboard Layout . . . . .	15
1.19	34 Keymap Library / Function Reference . . . . .	19

# Chapter 1

## Libraries

### 1.1 Amiga® RKM Libraries: 34 Keymap Library

Amiga computers are sold internationally with a variety of local keyboards which match the standards of particular countries. All Amigas have keyboards which are physically similar, and keys which output the same low-level raw key code for any particular physical key. However, in different countries, the keycaps of the keys may contain different letters or symbols. Since the physical position of a key determines the raw key code that it generates, raw key codes are not internationally compatible. For instance, on the German keyboard, the Y and Z keys are swapped when compared to the USA keyboard. The second key on the fifth row will generate the same raw key code on all Amiga keyboards, but should be decoded as a Z on a US keyboard and as a Y on a German keyboard.

The Amiga uses the ECMA-94 Latin1 International 8-bit character set, and can map raw key codes to any desired ANSI character value, string, or escape sequence. This allows national keyboards to be supported by using keymaps. A keymap is a file which describes what character or string is tied to what key code. Generally, the user's startup-sequence will set a system default keymap that is correct for the user's keyboard. The console.device translates the raw key codes into the correct characters based on the installed keymap. This includes the translation of special deadkey sequential key combinations to produce accented international characters.

Programs which perform keyboard input using the console.device, CON:, RAW:, or Intuition VANILLAKEY, will receive the correct ASCII values for a user's keycaps, based on their keymap. But some applications may require custom keymaps, or may need to perform their own translation between raw key codes and ANSI characters. In this chapter, the term ANSI refers to standard 8-bit character definitions which include printable ASCII characters, special characters, and escape sequences.

Until V37, keymapping commands were only available in the console.device. Keymap.library is a new library in Release 2 (V37). It offers the some of the keymap commands of the console.device, enabling applications to inquire after the default keymap and map key codes to ANSI characters. It also provides the ability to map ANSI characters back into raw codes. Unlike the console.device however, it can not be used to select a keymap for only one application, i.e., one console window.

As a prelude to the following material, note that the Amiga keyboard transmits raw key information to the computer in the form of a key position and a transition. Raw key positions range from hexadecimal 00 to 7F. When a key is released, its raw key position, plus hexadecimal 80, is transmitted.

Keymap Functions      Keyboard Layout      Function Reference

1.2    34 Keymap Library / Keymap Functions

Table 34-1: Keymap Library Functions

AskKeyMapDefault()	Ask for a pointer to current default keymap
MapANSI()	Encode an ANSI string into key codes
MapRawKey()	Decode a raw key input event to an ANSI string
SetKeyMapDefault()	Set the current default keymap for the system

Table 34-2: Console Device Keymap Commands

CD_ASKKEYMAP	Ask for the current console's keymap
CD_SETKEYMAP	Set the current console's keymap
CD_ASKDEFAULTKEYMAP*	Set the current default keymap
CD_SETDEFAULTKEYMAP**	Ask for a pointer to current default keymap
* Obsolete - use AskKeyMapDefault()	
** Obsolete - use SetKeyMapDefault()	

All of these commands deal with a set of pointers to key translation arrays, known as a KeyMap structure. The KeyMap structure is defined in <devices/keymap.h> and is shown below.

```
struct KeyMap
{
    UBYTE *km_LoKeyMapTypes;
    ULONG *km_LoKeyMap;
    UBYTE *km_LoCapsable;
    UBYTE *km_LoRepeatable;
    UBYTE *km_HiKeyMapTypes;
    ULONG *km_HiKeyMap;
    UBYTE *km_HiCapsable;
    UBYTE *km_HiRepeatable;
};
```

The KeyMap structure contains pointers to arrays which define the ANSI character or string that should be produced when a key or a combination of keys are pressed. For example, a keymap might specify that the key with raw value hex 20 should produce the ANSI character "a", and if the Shift key is being held it should produce the character "A".

- Asking For the Default Keymap
- Setting the Default Keymap
- Accessing the Keymap For the Current Console
- Mapping Key Codes To ANSI Strings
- Mapping ANSI Strings To Key Codes
- Details Of the Keymap Structure
- Key Map Standards
- Dead-Class Keys
- Double-Dead Keys

### 1.3 34 / Keymap Functions / Asking For the Default Keymap

The AskKeyMapDefault() returns a pointer to the current default keymap. To use the mapping functions in keymap.library it is normally not necessary to call this function. They accept NULL as equivalent to 'use default keymap' and will call this function for you. You can use this pointer for example to cache the system default in order to temporarily change the keymap your applications uses, or find the keymap in the keymap.resource list of loaded keymaps. You should never change the system wide default keymap unless the user asks you to do so; since the Amiga is a multitasking system, changing the keymap could interfere with the behaviour of other applications.

### 1.4 34 / Keymap Functions / Setting the Default Keymap

The system default keymap can be set with the SetKeyMapDefault() function. This function takes a pointer to a loaded keymap. In general, this function should never be used by an application unless the application is a system Preferences editor, or an application that takes over the system. Normal applications should instead attach a console.device unit to their own Intuition window (see the Devices volume), and use the console.device command CD\_SETKEYMAP to set a keymap only for their own console.

When making a keymap the system default, first check whether the keymap has been loaded previously by checking the keymap list of the keymap.resource. If it has not been loaded already, it can be loaded from devs:Keymaps, and added to the keymap list of keymap.resource. This will ensure that other applications which may want the keymap will not have to load a second instance. Once made the default, the keymap can never be safely removed from memory, even after if it is no longer the default, since other applications may still have and use a pointer to it.

### 1.5 34 / Keymap Functions / Accessing the Keymap For the Current Console

The function `AskKeyMap()` shown below does not return a pointer to a table of pointers to currently assigned key mapping. Instead, it copies the current set of pointers to a user-designated area of memory. `AskKeyMap()` returns a `TRUE` or `FALSE` value that says whether or not the function succeeded.

The function `SetKeyMap()`, also shown below, copies the designated key map data structure to the console device. Thus this routine is complementary to `AskKeymap()` in that it can restore an original key mapping as well as establish a new one.

`Ask/SetKeyMap()` functions.

-----  
 These functions assume that you have already opened the `console.device` and that request is a valid `IOStdReq` structure for the newly opened console. These functions are not part of the `keymap.library`, nor of the `console.device`. These merely demonstrate `CD_ASKKEYMAP` and `CD_SETKEYMAP` which are `console.device` commands.

```
/* These functions require that you have created a port and an IO request,
 * and have opened the console device as shown in the Console Device
 * chapter of the Devices volume of this manual set.
 */
```

```
#include <devices/keymap.h>
```

```
BOOL AskKeyMap(struct IOStdReq *request, struct KeyMap *keymap)
{
    request->io_Command = CD_ASKKEYMAP;
    request->io_Length = sizeof(struct KeyMap);
    request->io_Data = (APTR)keymap; /* where to put it */
    DoIO(request);
    if(request->io_Error) return(FALSE);
    else return(TRUE); /* if no error, it worked. */
}
```

```
BOOL SetKeyMap(struct IOStdReq *request, struct KeyMap *keymap)
{
    request->io_Command = CD_SETKEYMAP;
    request->io_Length = sizeof(struct KeyMap);
    request->io_Data = (APTR)keymap; /* where to get it */
    DoIO(request);
    if(request->io_Error) return(FALSE);
    else return(TRUE); /* if no error, it worked. */
}
```

## 1.6 34 / Keymap Functions / Mapping Key Codes To ANSI Strings

`MapRawKey()` is converts raw key codes to ANSI characters based on a default or supplied keymap.

```
WORD MapRawKey(struct InputEvent *inputevent, UBYTE *buffer,
                WORD bufferlength, struct Keymap *keymap);
```

---

MapRawKey() takes an IECLASS\_RAWKEY inputevent, which may be chained, and converts the key codes to ANSI characters which are placed in the specified buffer. If the buffer would overflow, for example because a longer string is attached to a key, -1 will be returned. If no error occurred, MapRawKey() will return the number of bytes written in the buffer. The keymap argument can be set to NULL if the default keymap is to be used for translation, or can be a pointer to a specific KeyMap structure.

The following example shows how to implement the MapRawKey() function.

maprawkey.c

## 1.7 34 / Keymap Functions / Mapping ANSI Strings To Key Codes

The MapANSI() function translates ANSI strings into raw key codes, complete with qualifiers and (double) dead keys, based on a default or supplied keymap.

```
LONG MapANSI(UBYTE *string, LONG stringlength, UBYTE *buffer,
             LONG bufferlength, struct KeyMap *keymap);
```

The string argument is a pointer to an ANSI string, of length stringlength. The buffer argument is a pointer to the memory block where the translated key codes will be placed. The length of this buffer must be indicated in WORDs since each translation will occupy one byte for the key code and one for the qualifier. Since one ANSI character can be translated to two dead keys and one key, the buffer must be at least 3 WORDs per character in the string to be translated. The keymap argument can be set to NULL if the default keymap is to be used, or can be a pointer to a KeyMap structure. Upon return, the function will indicate how many key code/qualifier combinations are placed in the buffer or a negative number in case an error occurred. If zero is returned, the character could not be translated.

The following example shows the usage of MapANSI() and demonstrates how returned key codes can be processed.

mapansi.c

## 1.8 34 / Keymap Functions / Details Of the Keymap Structure

A KeyMap structure contains pointers to arrays which determine the translation from raw key codes to ANSI characters.

```
struct KeyMap
{
    UBYTE *km_LoKeyMapTypes;
    ULONG *km_LoKeyMap;
    UBYTE *km_LoCapsable;
    UBYTE *km_LoRepeatable;
    UBYTE *km_HiKeyMapTypes;
```

---



```
ULONG *km_HiKeyMap;
UBYTE *km_HiCapsable;
UBYTE *km_HiRepeatable;
};
```

LoKeyMap and HighKeyMap	String Output Keys
LoKeyMapTypes and HiKeyMapTypes	Capsable Bit Tables
More About Qualifiers	Repeatable Bit Tables

1.9 34 // Details Of the Keymap Structure / LoKeyMap and HighKeyMap

The low key map provides translation of the key values from hex 00-3F; the high key map provides translation of key values from hex 40-7F. Key values from hex 68-7F are not used by the existing keyboards, but this may change in the future. A raw key value (hex 00-7F) plus hex 80 is the release of that key. If you need to check for raw key releases do it like this:

```
if (keyvalue & 0x80)      { /* do key up processing */ }
else                     { /* do key down processing */ }
```

Raw output from the keyboard for the low key map does not include the space bar, Tab, Alt, Ctrl, arrow keys, and several other keys.

Table 34-3: High Key Map Hex Values

Key Number	Keycap Legend or Function	Key Number	Keycap Legend or Function
40	Space	50-59	Function keys F1-F10
41	Backspace	5A-5E	Numeric Pad characters
42	Tab	5F	Help
43	Enter	60	Left Shift
44	Return	61	Right Shift
45	Escape	62	Caps Lock
46	Delete	63	Control
4A	Numeric Pad character	64	Left Alt
4C	Cursor Up	65	Right Alt
4D	Cursor Down	66	Left Amiga
4E	Cursor Right	67	Right Amiga
4F	Cursor Left		

The keymap table for the low and high keymaps consists of 4-byte entries, one per hex key code. These entries are interpreted in one of three possible ways:

- \* As four separate bytes, specifying how the key is to be interpreted when pressed alone, with one qualifier, with another qualifier, or with both qualifiers (where a qualifier is one of three possible keys: Ctrl, Alt, or Shift).
- \* As a longword containing the address of a string descriptor, where a string of characters is to be output when this key is

pressed. If a string is to be output, any combination of qualifiers can affect the string that may be transmitted.

- \* As a longword containing the address of a dead-key descriptor, where additional data describe the character to be output when this key is pressed alone or with another dead key.

The keymap tables must be word aligned.

-----  
The keymap tables must begin aligned on a word boundary. Each entry is four bytes long, thereby maintaining word alignment throughout the table. This is necessary because some of the entries may be longword addresses and must be aligned properly for the 68000.

## 1.10 34 // Details Of Keymap Structure / LoKeyMapTypes and HiKeyMapTypes

The tables named km\_LoKeyMapTypes and km\_HiKeyMapTypes each contain one byte per raw key code. Each byte defines the type of entry that is found in the keymap table for that raw key code.

Possible key types are:

- \* Any of the qualifier groupings noted below
- \* KCF\_STRING + any combination of KCF\_SHIFT, KCF\_ALT, KCF\_CONTROL (or KC\_NOQUAL) if the result of pressing the key is to be a stream of bytes (and key-with-one-or-more-qualifiers is to be one or more alternate streams of bytes).

Any key can be made to output up to eight unique byte streams if KCF\_STRING is set in its keytype. The only limitation is that the total length of all of the strings assigned to a key must be within the "jump range" of a single byte increment. See the "String Output Keys" section below for more information.

- \* KCF\_DEAD + any combination of KCF\_SHIFT, KCF\_ALT, KCF\_CONTROL (or KC\_NOQUAL) if the key is a dead-class key and can thus modify or be modified by another dead-class key. See the "Dead-Class Keys" section below for more information.

The low keytype table covers the raw key codes from hex 00-3F and contains one byte per key code. Therefore this table contains 64 (decimal) bytes. The high keytype table covers the raw key codes from hex 40-7F and contains 64 (decimal) bytes.

## 1.11 34 // Details Of the Keymap Structure / More About Qualifiers

For keys such as the Return key or Esc key, the qualifiers specified in the keytypes table (up to two) are the qualifiers used to establish the response to the key. This is done as follows. In the keytypes table, the values listed for the key types are those listed for the qualifiers in

<devices/keymap.h> and <devices/keymap.i>. Specifically, these qualifier equates are:

```
KC_NOQUAL      0x00
KCF_SHIFT      0x01
KCF_ALT        0x02
KCF_CONTROL    0x04
KC_VANILLA     0x07
KCF_DOWNUP     0x08
KCF_STRING     0x40
```

As shown above, the qualifiers for the various types of keys occupy specific bit positions in the key types control byte. As you may have noticed, there are three possible qualifiers, but only a 4-byte space in the table for each key. This does not allow space to describe what the computer should output for all possible combinations of qualifiers. A solution exists, however, for "vanilla" keys, such as the alphabetic keys. Here is how that works.

Keys of type KC\_VANILLA use the 4 bytes to represent the data output for the key alone, Shifted key, Alt'ed key, and Shifted-and-Alt'ed key. Then for the Ctrl-key-plus-vanilla-key, use the code for the key alone with bits 6 and 5 set to 0.

```
The Vanilla Qualifier Does Not Mean Plain.
-----
The qualifier KC_VANILLA is equivalent to
KCF_SHIFT+KCF_ALT+KCF_CONTROL.
```

This table shows how to interpret the keymap for various combinations of the qualifier bits:

Table 34-4: Keymap Qualifier Bits

If Keytype is:	Then data at this position in the keytable is output when the key is pressed along with:			
-----	-----			
KC_NOQUAL	-	-	-	alone
KCF_SHIFT	-	-	Shift	alone
KCF_ALT	-	-	Alt	alone
KCF_CONTROL	-	-	Ctrl	alone
KCF_ALT+KCF_SHIFT	Shift+Alt	Alt	Shift	alone
KCF_CONTROL+KCF_ALT	Ctrl+Alt	Ctrl	Alt	alone
KCF_CONTROL+KCF_SHIFT	Ctrl+Shift	Ctrl	Shift	alone
KC_VANILLA	Shift+Alt	Alt	Shift	alone*

\*Special case--Ctrl key, when pressed with one of the alphabet keys and certain others, is to output key-alone value with the bits 6 and 5 set to zero.

1.12 34 // Details Of the Keymap Structure / String Output Keys

When a key is to output a string, the keymap table contains the address of a string descriptor in place of a 4-byte mapping of a key. Here is a partial table for a new high keymap table that contains only three entries thus far. The first two are for the space bar and the backspace key; the third is for the tab key, which is to output a string that says "[TAB]". An alternate string, "[SHIFTED-TAB]", is also to be output when a shifted TAB key is pressed.

```
newHiMapTypes:
    DC.B   KCF_ALT,KC_NOQUAL,      ;key 41
    DC.B   KCF_STRING+KCF_SHIFT,  ;key 42
    ...   ;(more)
newHiMap:
    DC.B   0,0,$A0,$20   ;key 40: space bar, and Alt-space bar
    DC.B   0,0,0,$08     ;key 41: Back Space key only
    DC.L   newkey42      ;key 42: new string definition to output for Tab
    ...   ;(more)
newkey42:
    DC.B   new42ue - new42us      ;length of the unshifted string
    DC.B   new42us - newkey42     ;number of bytes from start of
                                ;string descriptor to start of this string
    DC.B   new42se - new42ss      ;length of the shifted string
    DC.B   new42ss - newkey42     ;number of bytes from start of
                                ;string descriptor to start of this string
new42us:   DC.B                '[TAB]'
new42ue:
new42ss:   DC.B                '[SHIFTED-TAB]'
new42se:
```

The new high map table points to the string descriptor at address newkey42. The new high map types table says that there is one qualifier, which means that there are two strings in the key string descriptor.

Each string in the descriptor takes two bytes in this part of the table: the first byte is the length of the string, and the second byte is the distance from the start of the descriptor to the start of the string. Therefore, a single string (KCF\_STRING + KC\_NOQUAL) takes 2 bytes of string descriptor. If there is one qualifier, 4 bytes of descriptor are used. If there are two qualifiers, 8 bytes of descriptor are used. If there are 3 qualifiers, 16 bytes of descriptor are used. All strings start immediately following the string descriptor in that they are accessed as single-byte offsets from the start of the descriptor itself. Therefore, the distance from the start of the descriptor to the last string in the set (the one that uses the entire set of specified qualifiers) must start within 255 bytes of the descriptor address.

Because the length of the string is contained in a single byte, the length of any single string must be 255 bytes or less while also meeting the "reach" requirement. However, the console input buffer size limits the string output from any individual key to 32 bytes maximum.

The length of a keymap containing string descriptors and strings is variable and depends on the number and size of the strings that you provide.

## 1.13 34 // Details Of the Keymap Structure / Capsable Bit Tables

The vectors `km_LoCapsable` and `km_HiCapsable` each point to an array of 8 bytes that contain more information about the keytable entries. Specifically, if the Caps Lock key has been pressed (the Caps Lock LED is on) and if there is a bit on in that position in the capsable map, then this key will be treated as though the Shift key is now currently pressed. For example, in the default key mapping, the alphabetic keys are "capsable" but the punctuation keys are not. This allows you to set the Caps Lock key, just as on a normal typewriter, and get all capital letters. However, unlike a normal typewriter, you need not go out of Caps Lock to correctly type the punctuation symbols or numeric keys.

In the byte array, the bits that control this feature are numbered from the lowest bit in the byte, and from the lowest memory byte address to the highest. For example, the bit representing capsable status for the key that transmits raw code 00 is bit 0 in byte 0; for the key that transmits raw code 08 it is bit 0 in byte 1, and so on.

There are 64 bits (8 bytes) in each of the two capsable tables.

## 1.14 34 // Details Of the Keymap Structure / Repeatable Bit Tables

The vectors `km_LoRepeatable` and `km_HiRepeatable` each point to an array of 8 bytes that contain additional information about the keytable entries. A bit for each key indicates whether or not the specified key should repeat at the rate set by the Input Preferences program.

The bit positions correspond to those specified in the capsable bit table. If there is a 1 in a specific position, the key can repeat. There are 64 bits (8 bytes) in each of the two repeatable tables.

## 1.15 34 / Keymap Functions / Key Map Standards

Users and programs depend on certain predictable behaviors from all keyboards and keymaps. With the exception of dead-class keys (see "Dead-Class Keys" section), mapping of keys in the low key map should follow these general rules:

- \* When pressed alone, keys should transmit the ASCII equivalent of the unshifted letter or lower symbol on the keycap.
- \* When Shifted, keys should transmit the ASCII equivalent of the shifted letter or upper symbol printed on the keycap.
- \* When Alt'ed, keys should generally transmit the same character (or act as the same deadkey) as the Alt'ed key in the usual keymap.
- \* When pressed with CTRL alone, alphabetic keys should generally transmit their unshifted value but with bits 5 and 6 cleared. This allows keyboard typing of "control characters." For

example, the C key (normally value \$63) should transmit value \$03 (Ctrl-C) when Ctrl and C are pressed together.

The keys in the high key map (keys with raw key values \$40 and higher) are generally non-alphanumeric keys such as those used for editing (backspace, delete, cursor keys, etc.), and special Amiga keys such as the function and help keys. Keymaps should translate these keys to the same values or strings as those shown in table 34-6, ROM Default Key Mapping.

In addition to their normal unshifted and shifted values, the following translations are standard for particular qualified high keymap keys:

Key	Generates This Value	If Used with Qualifier, Generates This Value
---	-----	-----
Space	\$20	\$A0 with qualifier KCF_ALT
Return	\$0D	\$0A with qualifier KCF_CONTROL
Esc	\$1B	\$9B with qualifier KCF_ALT

## 1.16 34 / Keymap Functions / Dead-Class Keys

All of the national keymaps, including USA, contain dead-class keys. This term refers to keys that either modify or can themselves be modified by other dead-class keys. There are two types of dead-class keys: dead and deadable. A dead key is one which can modify certain keys pressed immediately following. For example, on the German keyboard there is a dead key marked with the grave accent (`). The dead key produces no console output, but when followed by (for instance) the A key, the combination will produce the a-grave (à) character (National Character Code \$E0). On the U.S. keyboard, Alt-G is the deadkey used to add the grave accent (`) to the next appropriate character typed. A deadable key is one that can be prefixed by a dead key. The A key in the previous example is a deadable key. Thus, a dead key can only affect the output of a deadable key.

For any key that is to have a dead-class function, whether dead or deadable, the qualifier KCF\_DEAD flag must be included in the entry for the key in the KeyMapTypes table. The KCF\_DEAD type may also be used in conjunction with the other qualifiers. Furthermore, the key's keymap table entry must contain the longword address of the key's dead-key descriptor data area in place of the usual 4 ASCII character mapping.

Below is an excerpt from the Amiga 1000 German key map which is referred to in the following discussion.

```
KMLowMapType:
  DC.B    KCF_DEAD+KC_VANILLA    ; aA (Key 20)
        ...                      ; (more...)
  DC.B    KCF_DEAD+KC_VANILLA    ; hH (Key 25)
        ...                      ; (more...)

KMLowMap:
  DC.L    key20                   ; a, A, ae, AE
        ...                      ; (more...)
  DC.L    key25                   ; h, H, dead ^
        ...                      ; (more...)
```

```

;----- possible dead keys
key25:
    DC.B    0,'h',0,'H'           ; h, H
    DC.B    DPF_DEAD,3,DPF_DEAD,3 ; dead ^, dead ^
    DC.B    0,$08,0,$08,0,$88,0,$88 ; control translation
    ...           ; (more...)
;----- deadable keys (modified by dead keys)
key20:
    ; a, A, ae, AE
    DC.B    DPF_MOD,key20u-key20 ; deadable flag, number of
    ; bytes from start of key20
    ; descriptor to start of un-
    ; shifted data
    DC.B    DPF_MOD,key20s-key20 ; deadable flag, number of
    ; bytes from start of key20
    ; descriptor to start of shift-
    ; ed data
    DC.B    0,$E6,0,$C6           ; null flags followed by rest
    DC.B    0,$01,0,$01,0,$81,0,$81 ; of values (ALT, CTRL...)
key20u:
    DC.B    'a',$E1,$E0,$E2,$E3,$E4 ; 'a' alone and characters to
    ; output when key alone is
    ; prefixed by a dead key
    DC.B    $E1,$E1,$E2,$E1,$E1,$E1 ; most recent is '
    DC.B    $E0,$E2,$E0,$E0,$E0,$E0 ; most recent is `
key20s:
    DC.B    'A',$C1,$C0,$C2,$C3,$C4 ; SHIFTeD 'a' and characters to
    ; output when SHIFTeD key is
    ; prefixed by a dead key
    DC.B    $C1,$C1,$C2,$C1,$C1,$C1 ; most recent is '
    DC.B    $C0,$C2,$C0,$C0,$C0,$C0 ; most recent is `

```

In the example, key 25 (the H key) is a dead key and key 20 (the A key) is a deadable key. Both keys use the addresses of their descriptor data areas as entries in the LoKeyMap table. The LoKeyMapTypes table says that there are four qualifiers for both: the requisite KCF\_DEAD, as well as KCF\_SHIFT, KCF\_ALT, and KCF\_CONTROL. The number of qualifiers determine length and arrangement of the descriptor data areas for each key. The next table shows how to interpret the KeyMapTypes for various combinations of the qualifier bits. For each possible position a pair of bytes is needed. The first byte in each pair tells how to interpret the second byte (more about this below).

Table 34-5: Dead Key Qualifier Bits

If type is:	Then the pair of bytes in this position in the dead-class key descriptor data is output when the key is pressed along with:									
	alone	-	-	-	-	-	-	-	-	-
NOQUAL	alone	-	-	-	-	-	-	-	-	-
A	alone	A	-	-	-	-	-	-	-	-
C	alone	C	-	-	-	-	-	-	-	-

S	alone	S	-	-	-	-	-	-	
-----	-----	---	---	-----	---	-----	-----	-----	
A+C	alone	A	C	A+C	-	-	-	-	
-----	-----	---	---	-----	---	-----	-----	-----	
A+S	alone	S	A	A+S	-	-	-	-	
-----	-----	---	---	-----	---	-----	-----	-----	
C+S	alone	S	C	C+S	-	-	-	-	
-----	-----	---	---	-----	---	-----	-----	-----	
S+A+C (VANILLA)	alone	S	A	S+A	C	C+S	C+A	C+S+A	
-----	-----	---	---	-----	---	-----	-----	-----	
The abbreviations A, C, S stand for KCF_ALT, KCF_CONTROL, and KCF_SHIFT, respectively. Also note that the ordering is reversed from that in the normal KeyMap table.									

Because keys 20 and 25 each use three qualifier bits (not including KCF\_DEAD), according to the table there must be 8 pairs of data, arranged as shown. Had only KCF\_ALT been set, for instance, (not including KCF\_DEAD), just two pairs would have been needed.

As mentioned earlier, the first byte of each data pair in the descriptor data area specifies how to interpret the second byte. There are three possible values: 0, DPF\_DEAD and DPF\_MOD. In the Amiga 1000 German keymap listed above, DPF\_DEAD appears in the data for key 25, while DPF\_MOD is used for key 20. It is the use of these flags that determines whether a dead-class key has dead or deadable function. A value of zero causes the unrestricted output of the following byte.

If the flag byte is DPF\_DEAD, then that particular key combination (determined by the placement of the pair of bytes in the data table) is dead and will modify the output of the next key pressed (if deadable). How it modifies is controlled by the second byte of the pair which is used as an index into part(s) of the data area for ALL the deadable (DPF\_MOD set) keys.

Before going further, an understanding of the structure of a descriptor data area wherein DPF\_MOD is set for one (or more) of its members is necessary. Referring to the example, we see that DPF\_MOD is set for the first and second pairs of bytes. According to its LoKeyMapTypes entry, and using table 34-5 (Dead Key Qualifier Bits) as a guide, these pairs represent the alone and SHIFTEd values for the key. When DPF\_MOD is set, the byte immediately following the flag must be the offset from the start of the key's descriptor data area to the start of a table of bytes describing the characters to output when this key combination is preceded by any dead keys. This is where the index mentioned above comes in. The value of the index from a prefixing dead key is used to determine which of the bytes from the deadable keys special table to output. The byte in the index+1 position is sent out. (The very first byte is the value to output if the key was not prefixed by a dead key.) Thus, if Alt-H is pressed (dead) and then Shift-A, an 'a' with a circumflex (^) accent will be output. This is because:

- \* The byte pair for the ALT position of the H key (key 25) is DPF\_DEAD,3 so the index is 3.



- \* The byte pair for the SHIFT position of the A key (key 20) is DPF\_MOD, key20s-key20, so we refer to the table-of-bytes at key20s.
- \* The third+1 byte of the table-of-bytes is \$C2, an 'a' character.

A Note About Table Size.

-----

The number of bytes in the table-of-bytes for all deadable keys must be equal to the highest index value of all dead keys plus 1.

## 1.17 34 / Keymap Functions / Double-Dead Keys

Double-dead keys are an extension of the dead-class keys explained above. Unlike normal dead keys wherein one dead key of type DPF\_DEAD can modify a second of type DPF\_MOD, double-dead keys employ two consecutive keys of type DPF\_DEAD to together modify a third of type DPF\_MOD.

For example, the key on the German keyboard labeled with single quotes ( ' ) is a double-dead key. When this key is pressed alone and then pressed again shifted, there is no output. But when followed by an appropriate third key, for example the A key, the three keypresses combine to produce an 'a' with a circumflex (^) accent (character code \$E2). Thus the double-dead pair qualify the output of the A key.

The system always keeps the last two down key codes for possible further translation. If they are both of type DPF\_DEAD and the key immediately following is DPF\_MOD then the two are used to form an index into the (third) key's translation table as follows:

In addition to the index found after the DPF\_DEAD qualifier in a normal dead key, a second factor is included in the high nibble of double-dead keys (it is shifted into place with DP\_2DFACSHIFT). Its value equals the total number of dead key types + 1 in the keymap. This second index also serves as an identifying flag to the system that two dead keys can be significant.

When a key of type DPF\_MOD is pressed, the system checks the two key codes which preceded the current one. If they were both DPF\_DEAD then the most recent of the two is checked for the double-dead index/flag. If it is found then a new index is formed by multiplying the value in lower nibble with that in the upper. Then, the lower nibble of the least recent DPF\_DEAD key is added in to form the final offset.

Finally, this last value is used as an index into the translation table of the current, DPF\_MOD, key.

The translation table of all deadable (DPF\_MOD) keys has [number of dead key types + 1] \* [number of double dead key types + 1] entries, arranged in [number of double dead key types + 1] rows of [number of dead key types + 1] entries. This is because as indices are assigned for dead keys in the keymap, those that are double dead keys are assigned the lower numbers.

Following is a code fragment from the German (d) keymap source:

```

key0C:
    DC.B    DPF_DEAD,1+(6<<DP_2DFACSHIFT)    ; dead '
    DC.B    DPF_DEAD,2+(6<<DP_2DFACSHIFT)    ; dead `
    DC.B    0,'=',0,'+'                      ; =, +
key20:
    ; a, A, ae, AE
    DC.B    DPF_MOD,key20u-key20,DPF_MOD,key20s-key20
    DC.B    0,$E6,0,$C6
    DC.B    0,$01,0,$01,0,$81,0,$81 ; control translation
key20u:
    DC.B    'a',$E1,$E0,$E2,$E3,$E4
    DC.B    $E1,$E1,$E2,$E1,$E1,$E1 ; most recent is '
    DC.B    $E0,$E2,$E0,$E0,$E0,$E0 ; most recent is `
key20s:
    DC.B    'A',$C1,$C0,$C2,$C3,$C4
    DC.B    $C1,$C1,$C2,$C1,$C1,$C1 ; most recent is '
    DC.B    $C0,$C2,$C0,$C0,$C0,$C0 ; most recent is `

```

Raw key0C, the German single quotes ( `` ) key, is a double dead key. Pressing this key alone, then again while the shift key is down will produce no output but will form a double-dead qualifier. The output of key20 (A), a deadable key, will consequently be modified, producing an "a" with a circumflex (^) accent. The mechanics are as follows:

- \* When key0C is pressed alone the DPF\_DEAD of the first byte pair in the key's table indicates that the key is dead. The second byte is then held by the system.
- \* Next, when key0C is pressed again, this time with the Shift key down, the DPF\_DEAD of the second byte pair (recall that the second pair is used because of the SHIFT qualifier) again indicates the key is a dead key. The second byte of this pair is also held by the system.
- \* Finally, when the A key is pressed the system recalls the latter of the two bytes it has saved. The upper nibble, \$6, is multiplied by the lower nibble, \$2. The result, \$0C, is then added to the lower nibble of the earlier of the two saved bytes, \$1. This new value, \$0D, is used as an index into the (unshifted) translation table of key20. The character at position \$0D is character \$E2, an 'a' with a circumflex (^) accent.

Note About Double Dead Keys.

-----

If only one double-dead key is pressed before a deadable key then the output is the same as if the double-dead were a normal dead key. If shifted key0C is pressed on the German keyboard and then immediately followed by key20, the output produced is character \$E0, `à'. As before, the upper nibble is multiplied with the lower, resulting in \$0C. But because there was no second dead-key, this product is used as the final index.

## 1.18 34 Keymap Library / Keyboard Layout

The keys with key codes \$2B and \$30 in the following keyboard diagrams are keys which are present on some national Amiga keyboards.

Figure 34-1: Amiga 1000 Keyboard Showing Key Codes in Hex

Figure 34-2: Amiga 500/2000/3000 Keyboard Showing Key Codes in Hex

The default values given above correspond to the values the console device will return when these keys are pressed with the keycaps as shipped with the standard American keyboard.

Table 34-6: ROM Default (USA0) and USA1 Console Key Mapping

Raw Key Number	Keycap Legend	Unshifted Default Value	Shifted Default Value
00	` ~	` (Accent grave)	~ (tilde)
01	1 !	1	!
02	2 @	2	@
03	3 #	3	#
04	4 \$	4	\$
05	5 %	5	%
06	6 ^	6	^
07	7 &	7	&
08	8 *	8	*
09	9 (	9	(
0A	0 )	0	)
0B	- _	- (Hyphen)	_ (Underscore)
0C	= +	=	+
0D			
0E		(undefined)	
0F	0	0	0 (Numeric pad)
10	Q	q	Q
11	W	w	W
12	E	e	E
13	R	r	R
14	T	t	T
15	Y	y	Y
16	U	u	U
17	I	i	I
18	O	o	O
19	P	p	P
1A	[ {	[	{
1B	] }	]	}
1C		(undefined)	
1D	1	1	1 (Numeric pad)
1E	2	2	2 (Numeric pad)
1F	3	3	3 (Numeric pad)
20	A	a	A
21	S	s	S
22	D	d	D
23	F	f	F
24	G	g	G
25	H	h	H
26	J	j	J
27	K	k	K
28	L	l	L

29	; :	;	:
2A	' "	' (single quote)	"
2B		(not on most US keyboards)	
2C		(undefined)	
2D	4	4	4 (Numeric pad)
2E	5	5	5 (Numeric pad)
2F	6	6	6 (Numeric pad)
30		(not on most US keyboards)	
31	Z	z	Z
32	X	x	X
33	C	c	C
34	V	v	V
35	B	b	B
36	N	n	N
37	M	m	M
38	, <	, (comma)	<
39	. >	. (period)	>
3A	/ ?	/ ?	
3B		(undefined)	
3C	.	.	. (Numeric pad)
3D	7	7	7 (Numeric pad)
3E	8	8	8 (Numeric pad)
3F	9	9	9 (Numeric pad)
40	(Space bar)	20	20
41	Back Space	08	08
42	Tab	09	09
43	Enter	0D	0D (Numeric pad)
44	Return	0D	0D
45	Esc	1B	1B
46	Del	7F	7F
47		(undefined)	
48		(undefined)	
49		(undefined)	
4A	-	-	- (Numeric Pad)
4B		(undefined)	
4C	Up arrow	<CSI>A	<CSI>T
4D	Down arrow	<CSI>B	<CSI>S
4E	Forward arrow	<CSI>C	<CSI> A (note blank space after <CSI>)
4F	Backward arrow	<CSI>D	<CSI> @ (note blank space after <CSI>)
50	F1	<CSI>0~	<CSI>10~
51	F2	<CSI>1~	<CSI>11~
52	F3	<CSI>2~	<CSI>12~
53	F4	<CSI>3~	<CSI>13~
54	F5	<CSI>4~	<CSI>14~
55	F6	<CSI>5~	<CSI>15~
56	F7	<CSI>6~	<CSI>16~
57	F8	<CSI>7~	<CSI>17~
58	F9	<CSI>8~	<CSI>18~
59	F10	<CSI>9~	<CSI>19~
5A	(	(	( (usal Numeric pad)
5B	)	)	) (usal Numeric pad)
5C	/	/	/ (usal Numeric pad)
5D	*	*	* (usal Numeric pad)

5E	+	+	+ (usual Numeric pad)
5F	HELP	<CSI>?~	<CSI>?~

Raw Key Number	Function or Keycap Legend	
-----	-----	
60	Shift (left of space bar)	
61	Shift (right of space bar)	
62	Caps Lock	
63	Ctrl	
64	(Left) Alt	
65	(Right) Alt	
66	Amiga (left of space bar)	Left Amiga
67	Amiga (right of space bar)	Right Amiga
68	Left mouse button (not converted)	Inputs are only for the mouse connected to Intuition, (currently gameport one).
69	Right mouse button (not converted)	
6A	Middle mouse button (not converted)	
6B	(undefined)	
6C	(undefined)	
6D	(undefined)	
6E	(undefined)	
6F	(undefined)	
70-7F	(undefined)	
80-F8	Up transition (release or unpress key of one of the above keys) (80 for 00, F8 for 7F)	
F9	Last key code was bad (was sent in order to resynchronize)	
FA	Keyboard buffer overflow	
FB	(undefined, reserved for keyboard processor catastrophe)	
FC	Keyboard selftest failed	
FD	Power-up key stream start. Keys pressed or stuck at power-up will be sent between FD and FE.	
FE	Power-up key stream end	
FF	(undefined, reserved)	
FF	Mouse event, movement only, no button change (not converted)	

Notes about the preceding table:

- 1) "<CSI>" is the Control Sequence Introducer, value hex 9B.
- 2) "(undefined)" indicates that the current keyboard design should not generate this number. If you are using SetKeyMap() to change the key map, the entries for these numbers must still be included.
- 3) "(not converted)" refers to mouse button events. You must use the sequence "<CSI>2{" to inform the console driver that you wish to receive mouse events; otherwise these will not be transmitted.
- 4) "(RESERVED)" indicates that these key codes have been reserved for national keyboards. The \$2B code key will be between the double-quote (") and Return keys. The \$30 code key will be between

the Shift and Z keys.

+-----+   0   0   0   0   0   0   0   0   1   1   1   1   1   1   1   1     0   0   0   0   1   1   1   1   0   0   0   0   1   1   1   1     0   0   1   1   0   0   1   1   0   0   1   1   0   0   1   1     0   1   0   1   0   1   0   1   0   1   0   1   0   1   0   1                                       00  01  02  03  04  05  06  07  08  09  0a  0b  0c  0d  0e  0f  +-----+   0000 00                                     0001 10                                     0010 20   SP  !   "   #   \$   %   &   '   (   )   *   +   ,   -   .   /     0011 30   0   1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ?     0100 40   @   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O     0101 50   P   Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^   _     0110 60   `   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o     0111 70   p   q   r   s   t   u   v   w   x   y   z   {       }   ~       1000 80                                     1001 90                                     1010 a0   NBSP   ¡   ¢   £   ¤   ¥   ¨   ¤   ¤   ¤   ¤   ¤   ¤   ¤   ¤   ¤       SHY   ®   ¯       1011 b0   \textdegree{}   \ensuremath{\pm}   \$^2\$   \$^3\$   ´   \$\mathrm{\mu}\$   ¤   ¤       ¤   ¤   ¤   ¤   ¤   ¤   ¤   ¤   ¤   ¤   ¤   ¤   ¤   ¤   ¤     1100 c0   À   Á   Â   Ã   Ä   Å   Æ   Ç   È   É   Ê   Ë   Ì   Í   Î   Ï     1101 d0   Ð   Ñ   Ò   Ó   Ô   Õ   Ö   Ø   Ù   Ú   Û   Ü   Ý   Þ   ß     1110 e0   à   á   â   ã   ä   å   æ   ç   è   é   ê   ë   ì   í   î   ï     1111 f0   ð   ñ   ò   ó   ô   õ   ö   ø   ù   ú   û   ü   ý   þ   ÿ   +-----+															
---	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Figure 34-3: ECMA-94 Latin1 International 8-Bit Character Set

1.19 34 Keymap Library / Function Reference

The following chart gives a brief desription of the functions covered in this chapter. All of these functions require Release 2 or a later version of the Amiga operating system. See the Amiga ROM Kernel Reference Manual: Includes and Autodocs for details on each function call.

Table 34-7: Keymap Library Functions

Function	Description
AskKeyMapDefault()	Get a pointer to the current system default keymap
MapANSI()	Convert ANSI string to raw key events
MapRawKey()	Convert raw key events to ANSI
SetKeyMapDefault()	Set the system default keymap