BP30
Keyboard driver Specification

Edition 2006

Published by Neonseven s.r.l.,
Viale Stazione di Prosecco, 15

34010 Sgonico (Trieste) Italy

For questions on technology, delivery and prices please contact the Neonseven Offices in Italy Sgonico and Gorizia

Attention Please!

The information herein is given to describe certain components and shall not be considered as warranted characteristics.

Terms of delivery and rights to technical change reserved.

We hereby disclaim any and all warranties, including but not limited to warranties of non-infringement, regarding circuits, descriptions and charts stated herein.

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact Neonseven.

Neonseven technologies may only be used in life-support devices or systems with the express written approval of Neonseven, if a failure of such technologies can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

Table of Contents

1 Document Mission/Scope

1.1 Mission

This document contains a specification of the keyboard driver used in BP30 project. Such driver controls the keypad, the flip and the hook key; the headset detection is handled by the accessory driver.

1.2 Scope

This document is addressed to SW developers who need to either interface to or customize the Keyboard driver module.

2 List of Acronyms

Acronym	Meaning
GPIO	General Purpose Input Output
HISR	Higher Interrupt Service Routine
LISR	Lower Interrupt Service Routine
MMI	Man-Machine Interface
SDL	Specification and Description Language

3 Introduction

The purpose of the keyboard driver is to detect key press, key release (including the hook key), flip open and flip close and to forward this information to the MMI. A debounce algorithm is implemented to cope with the typical malfunction of mechanical switches such as keys and flip.

3.1 The E-GoldRadio keypad

The E-GoldRadio keypad interface is a 10-bit port with four input pins, six output pins and two addressable 16 bit interface registers, which are KBDINP for input pins and KBDOUT for output pins. Figure 3-1 shows a block diagram of the internal keypad port circuitry.

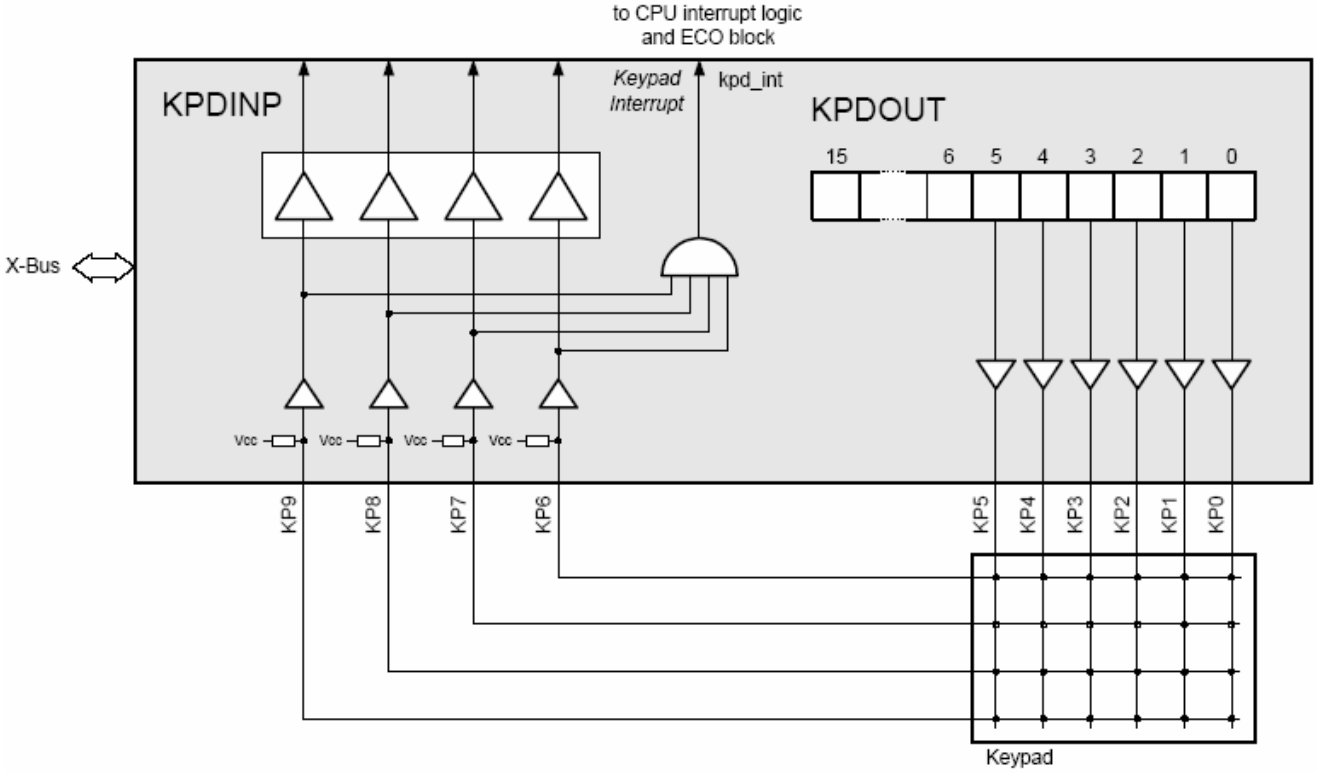

Figure 3-1

The levels at the input pins KP(9:6) are loaded into KPDINP with a read access. The levels of the output pins KP(5:0) are loaded into KPDOUT with a write access. Pressing a key directly generates an interrupt to the MCU, releasing this key generates an additional interrupt.
In order to check the key status, the keyboard driver loads KPDOUT with 0 to start monitoring the keypad. As long as no input port pin (KP9...6) is connected to any output port pin (KP5...0), the input pins are pulled high internally and a high level appears on the keypad interrupt input of the controller. If a key is pressed, both the corresponding keypad row and column are connected to each other and the corresponding KPDINP register bit is set to LOW. Due to the LOW level the kpd_int interrupt fires (active high, duration 2 X-Bus clock cycles) and is detected by the controller and the ECO block, which is then able to initiate an early wake-up procedure in case of sleep mode. Knowing the right row number the controller scans the six bits of KPDOUT by switching them from 1 to 0 step by step, therefore getting the right column number too.
When the release of the same key is detected, a second interrupt is generated and the keyboard matrix can be scanned again in the same manner.

According to E-GoldRadio documentation:

- There must be an instruction or NOP between a write to KPDOUT and a read from KPDINP to avoid pipeline effects.
- The interrupt to the CPU is generated even when the XBus clock is switched off.
- If the keypad port is not used the inputs are pulled high internally. KPDINP is not a real register: in fact it is a set of four parallel drivers (with enable) which connect the input pins with the internal bus. A read access will open these drivers.
- Pressing two or more keys at the same time leads to a short circuit of two output pins and therefore to a short circuit current. Reducing the short circuit current requires either external series resistors or a limitation of the short circuit time.

4 Architecture

The architecture of the keyboard driver is illustrated in Figure 4.1. The main control part is implemented as a finite state machine. It controls all key press and release actions and activates a HISR, which in turn sends this information to the MMI via SDL Signals. The state machine is triggered by three LISR, each handling a specific interrupt: KPDIC fires on each key press and release, T5INT generates a debounce delay and FEX7IC fires when opening the flip. Besides the “key press” and “key release” mails, the keyboard driver interface is limited to an initialization function called by the MMI on power up and a small set of test functions mainly for production.
Note1: The closing flip is not detected by interrupt because EX7IN detects only the interrupt on rising edge. The Flip Close status is detected by testing Px bit of input line (PCL_44). If Px=0 flip is close.The polling is performed each 1 second (216 frames) by using a Sw timer. (KEY_polling_tick called on a regular basis by the framer)
Note2: Flip detections is not performed on Goldfinch platform

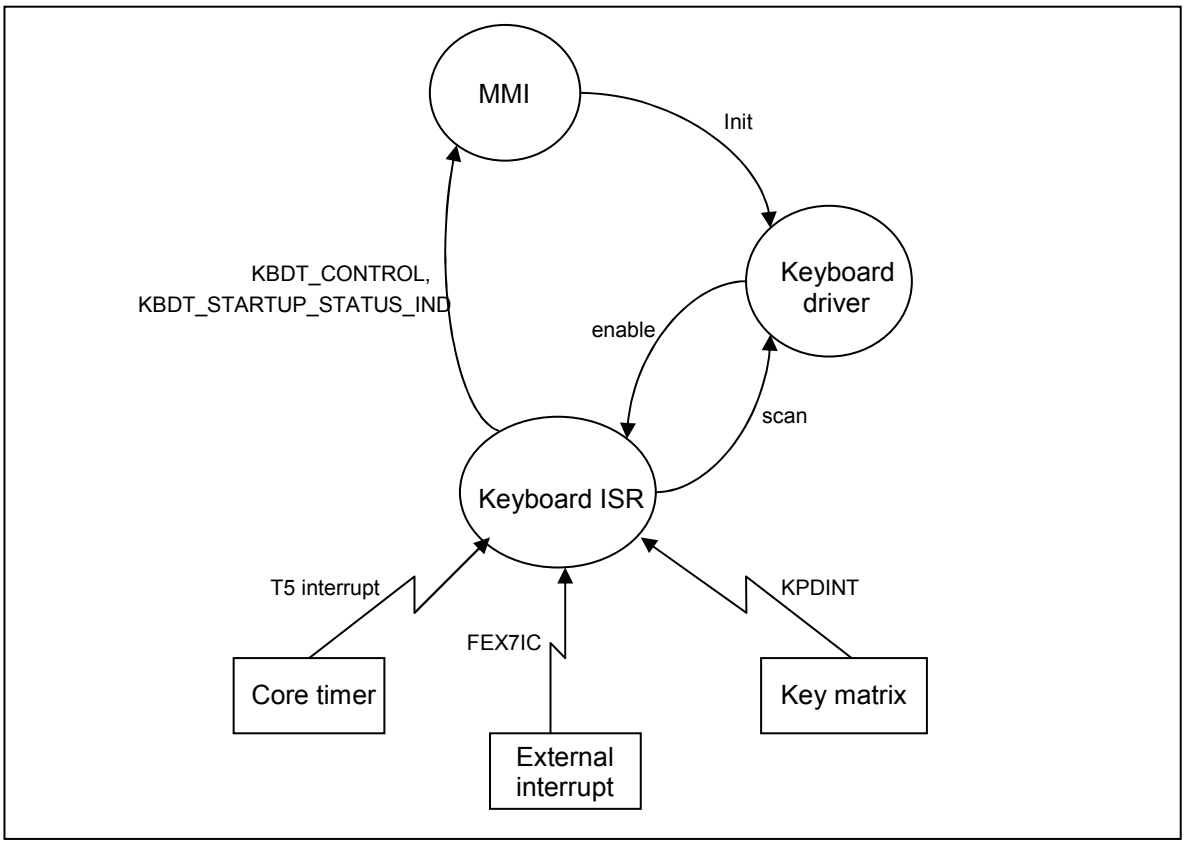

Figure 4-1

4.1 The keyboard matrix

The keyboard matrix contains 24 elements, identified by an index running from 0 to 23. Each element in the keyboard matrix can be associated to a key, named PBn (n=1,2,…) in the schematics. Table 4-1 contains the keyboard matrix with the associated keys. KEYIN3 column, even though empty, actually refers to the joystick.

	KEYIN0	KEYIN1	KEYIN2	KEYIN3
KEYOUT5	PB3	PB4	HOOK	PB9
KEYOUT4	PB15	PB5	PB2	RIGHT
KEYOUT3	PB11	PB1	PB14	UP
KEYOUT2	PB8	PB6	PB7	OK
KEYOUT1	PB17	PB18	PB13	LEFT
KEYOUT0	PB16	PB10	PB12	DOWN

Table 4-1

The keyboard hardware driver defines an array that maps the physical keys to their logical interpretation according to Table 4-2: left, right, up, down and ok refer to the joystick while headset represents the hook key.

INDEX	KEY	FUNCTION
0	PB16	*
1	PB17	7
2	PB8	4
3	PB11	1
4	PB15	SND
5	PB3	LSO
6	PB10	0
7	PB18	8
8	PB6	5
9	PB1	2
10	PB5	C
11	PB4	RSO
12	PB12	#
13	PB13	9
14	PB7	6
15	PB14	3
16	PB2	DEL
17		HOOK
18		DOWN
19		LEFT
20		OK
21		UP
22		RIGHT
23	PB9	FREE

Table 4-2

Two keys are handled in a special way.

- The Flip key SW1, which is not a part of the keyboard matrix, is directly connected to GPIO 44 configured as ALT-1 input; an interrupt will occur on rising edge and the GPIO output will toggle accordingly, allowing the detection of flip open event (flip close is handled by polling).
- The “ON-KEY”:
  - On BP30 platform (Globe6 Board) Keyin2 and Keyout4 are connected together (PB2) as soon as the ON-KEY is pressed or the RTC_OUT event is triggered (Alarm event)

- On Goldfinch platform: FEX7IC fires when the the ON-KEY is pressed or RTC_OUT event is triggered. (i.e as soon as the ON-KEY is pressed, VBat is connected to EX7IN, the rising edge is detected and the ISR KEY_on_off_lowisr is called.

The hook key, though not being part of the keyboard itself, is treated as a normal key because it's connected between KEYIN2 and KEYOUT5. Anyway the headset detection is responsibility of the accessory driver.

5 Functional description

The main control part of the keyboard driver (Key main control) is implemented as a state machine running in the context of a LISR.
Key main control is activated by three different interrupt sources: Keypad interrupt (key press/release), Core timer interrupt (debouncing) and an external interrupt (flip key on BP30, ON-KEY on Goldfinch), all sharing the same priority level (ILVL). The group level order is described in Table 5-1.

Interrupt type	ILVL	GLVL
KPDINT	4	1
hardware timer	4	2
External	4	0

Table 5-1

5.1 Key main control

Key main control is a state machine consisting of three different states: key_disable, key_pressed and key_released. Figure 5-1 shows the associated state chart. Note that the flip key is not represented as it does not trigger any state change.

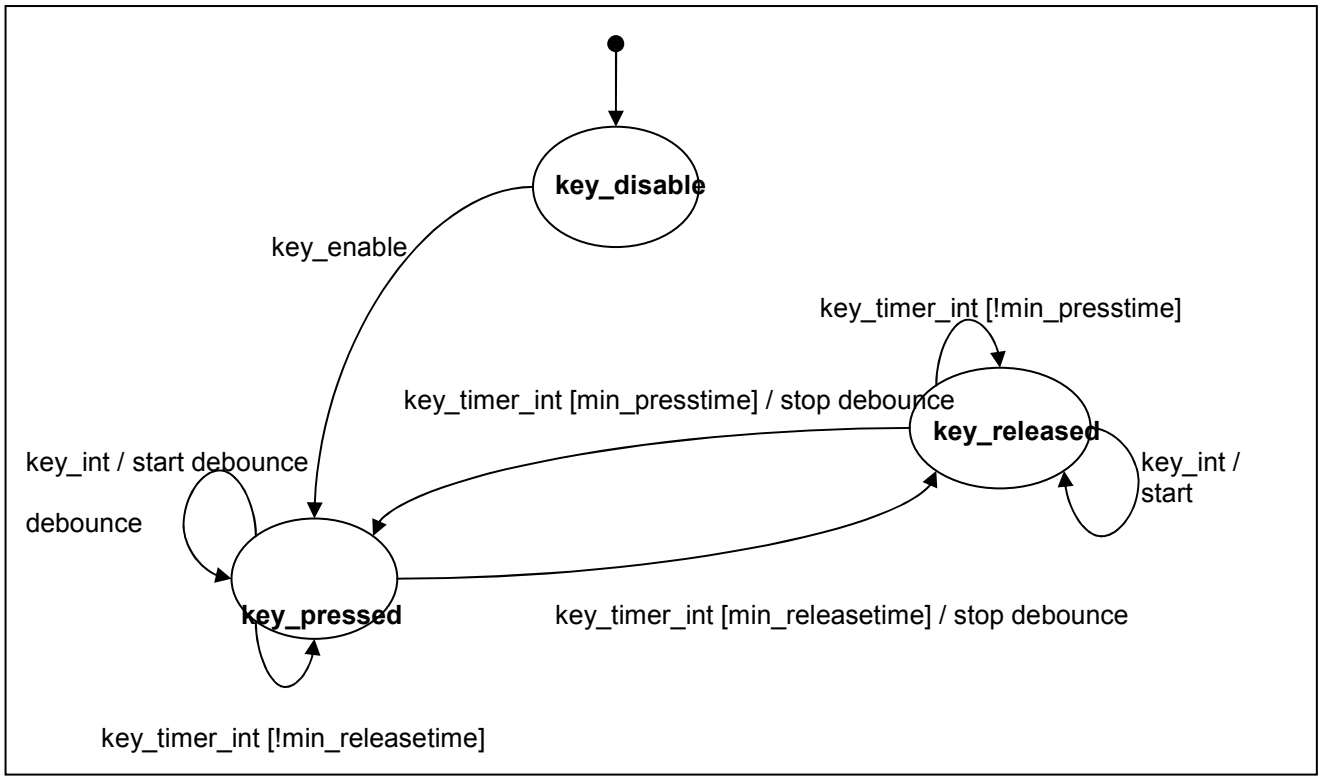

Figure 5-1

Disable:
This is the default state on startup. During initialization the MMI calls KEY_init function, triggering the enable event. Key main control scans the keyboard to check whether the “ON-KEY” is pressed then it activates the HISR routine, which sends the KBDT_STARTUP_STATUS_IND SDL signal to the MMI containing either a “KEY_DEL” or a “KEY_NONE” pressed indication. In the first case the T5 hardware timer interrupt is enabled to start the debouncing procedure and Key main control enters the state Key pressed, otherwise a transition to the Key released state occurs.

Key released:
In this state all keys are released. Possible events are:

1) *Key press*: a keyboard interrupt is generated and Key main control is triggered by a key_int event. The KPDIC and FEX7IC interrupts are disabled and the keyboard matrix is scanned to determine which key was pressed. If a valid key is still pressed then the T5 interrupt is enabled and the debouncing procedure begins. On timer expiry Key main control is called with a key_timer_int event and the keyboard matrix is scanned again, checking if any change took place. At the end of the debouncing period (after MIN_PRESSTIME) and only if the keyboard matrix maintained its state, the HISR routine is activated: key pressed mail (KBDT_CONTROL SDL signal) is sent to the MMI and Key main control enters the Key pressed state. If either the scan determines that no valid key is pressed or if the keyboard matrix status changed during debouncing then no state transition occurs and no mail is sent to the MMI; anyway the keyboard and external interrupt are re-enabled.
2) *FEX7IC interrupt*:
   - On BP30 a deboucing procedure is started via the T5 timer, during which Key main control checks if the flip was actually opened or closed. If its status is maintained during the debounce period and if it is different from the previous one then the HISR routine is activated and a “KEY_FLIP_OPEN” / “KEY_FLIP_CLOSED” mail is sent to the MMI via the KBDT_CONTROL SDL signal. In any case Key

main control state is not affected (it remains Key released). After debouncing the keyboard and FEX7IC interrupts are enabled again.

- On Goldfinch FEX7IC is used to detect the ON-KEY. As soon as the interrupt is generated the Key main control is triggered by a key_int event. Handled as in the Key released state case 1.

Key pressed:
This state is entered when a key is pressed. There are 2 possible events:

1) *Key release*: a keyboard interrupt is generated and Key main control is triggered by a key_int event. The KPDIC and FEX7IC interrupts are disabled and the keyboard matrix is scanned to determine which key was released. The debouncing procedure described above is executed and after MIN_RELEASETIME Key main control activates the HISR routine, which sends a key released mail via the KBDT_CONTROL SDL signal to the MMI. When no key is pressed any more a transition to the Released state occurs.At the end the keyboard and external interrupt are re-enabled.
2) *FEX7IC interrupt*: handled as in the Key pressed state.

5.2 Interrupt service routines

Keypad_HighISR:
Handles communication with the MMI via SDL signals; it is normally waiting for a fast semaphore activated by Key main control, which in turn is triggered by one of the lower ISRs. Kypad_HISR sends a startup signal after keypad initialization and then it notifies the MMI of each key press and each key release. The keyboard driver has no memory of previous keyboard status; therefore it is responsibility of the MMI to remember which keys are currently pressed. The HISR is necessary in order to release the LISRs from heavy and time-consuming activities such as dynamic memory allocation and signal processing.

KEY_keypad_lowisr:
Handles keyboard interrupts caused by either a key press or a key release; disables KPDIC and triggers Key main control with a key_int event, which starts scanning and debouncing.

KEY_timer5_lowisr:
Handles the debouncing procedure; reloads timer T5 with the proper period and calls Key main control with a key_timer_int event.

KEY_flip_lowisr: (Only BP30)
Triggered by the flip interrupt, which fires each time the flip is either opened or closed; calls Key main control with a key_flip_int event, which starts the associated debouncing procedure and (on successful completion) the notification to the MMI.

KEY_on_off_lowisr. (Only on Goldfinch)
Handles keyboard interrupts caused by ON-KEY press or RTC_OUT event; disables KPDIC and FEX7IC and triggers Key main control with a key_int event, which starts scanning and debouncing.

5.3 Multiple keys detection

The keyboard driver supports both single and multiple keys press/release detection.

- *Single mode*: each time a key is pressed the driver informs the MMI via an SDL Signal containing the pressed key code; such key press message is always followed by a key release message containing the fixed release code 0x22.

- *Multi mode*: for each key press the driver notifies the MMI via an SDL Signal with the pressed key code; for each key release the driver sends an SDL Signal with the fixed code 0x22 plus the released key code. Now

a key press message is not necessarily followed by a key release message; if more keys are simultaneously pressed or release several mails are sent to the MMI, each related to a single key.

In order to support multiple keys detection the driver keeps scans the whole keyboard matrix, instead of stopping after the first key press detection as in single mode. After each scan the result is stored in key_mail_matrix; if such matrix remains invariant during the debounce period then it is compared to its last invariant value stored into key_old_mail_matrix, which is updated each time a mail is sent. If the two matrices differ then the key_press_flag is set and the differences just found are saved into a key_mail_array, the old_mail_matrix is updated and a signal to the HISR is sent. The HISR retrieves the value stored into key_mail_array, then prepares and sends the corresponding mails to the MMI. When a key press is detected in the LISR (after debounce) the T5 debounce timer is stopped and is re-started only after the HISR has sent all the mails pending in the key_mail_array. This ensures that the LISR cannot change the key_mail_array while the HISR is sending SDL Signals corresponding to the key_mail_array. As long as at least one key is pressed, the driver will remain in key_pressed state, the key_released state can be therefore interpreted as an “all key released” state.
As already mentioned above the keyboard driver sends an SDL Signal for each key press and another one for each key release, without any additional information about simultaneous key press; it is responsibility of the MMI to remember the current keyboard status (i.e. which keys are currently pressed) based on the sequence of press and release mails received from the driver.

Note: not all he possible combinations of multiple keys simultaneously pressed are correctly detected: refer to Annex 1 for a detailed description of the limitations of multi mode and of the new scan algorithm implemented in order to avoid short circuits among output pins.

5.4 Long Keypress detections

Long key detection is handled by APOXI, it is not handled in Key driver.
When a key is pressed the interrupt is generated, KPDIC is disable, scanning procedure is performed, the key is detected and the key_pressed indications is sent to MMI. If at the end of the key detection procedure the Key is still pressed, the interrupt is re-generated, the key is re-detected but no indication is sent to MMI because the keyboard matrix is unchanged.
As soon as the Key is released a new interrupt is generated and the Key_release message is sent to MMI.

5.5 Hook key detection

The hook key, even though mapped onto the keyboard matrix just as any other key, needs a special handling because when it’s pressed it pulls down the corresponding KEYIN row, which other keys do not. Therefore at the beginning of the keypad scan all KPDOUT bits are set to high; if the KEYIN2 row is detected as low with all outputs set to high then the hook key must be pressed, therefore the scan is interrupted and an SDL Signal is sent to the MMI. In this situation it’s not possible to detect the pressure of any other key in KEYIN2 row.

5.6 Interface specification

The keyboard driver interface is based on function calls from the MMI to the driver and on SDL Signals from the driver to the MMI.

5.6.1 *SDL Signals*

5.6.1.1 *KBDT_CONTROL*

Description: notifies the MMI of either a key press or a key release, detected by a KPDIC and confirmed after a successful debouncing.
Parameters: they differ according to both the keyboard driver mode (single/multi) and the message meaning (key press/release):

- Param1: length (always 1)
- Param2: key code for a key press, 0x22 for a key release
- Param3: key code for a key release in multi mode, dummy (always 0) otherwise

5.6.1.2 KBDT_STARTUP_STATUS_IND

Description: notifies the MMI of the keyboard driver start up, including information about the “ON-KEY” status (either pressed or released).
Parameters:

- Param1: active flag, true if the “ON-KEY” is pressed on start up
- Param2: length (always 1)
- Param3: key code
- Param4: dummy (always 0)

5.6.2 *Operative interface*

5.6.2.1 *KEY_init*

Prototype: void KEY_init (void)
Parameters: None.
Functional description: KEY_init is called on power up when Keypad_HighISR process is created. KEY_init performs the required initialization of the keypad registers, interrupt registers and global variables. Then it triggers the first transition of the keyboard driver state machine, by calling Key main control with a key_enable event. Key main control scans the keyboard then sends a start-up indication to the MMI, containing the status of the “ON-KEY”: in the normal start-up procedure the “ON-KEY” is expected to be pressed. After initialization the keyboard driver is ready to detect key events.

5.6.2.2 *KEY_force_power_up*

Prototype: void KEY_force_power_up (void)
Parameters: None.
Functional description: Called during initialization, forces normal power up by overwriting key with "ON-KEY" and prepares 0.5 sec delay before handling normal release procedure.

5.6.2.3 *KEY_get_flip_status*

Prototype: int KEY_get_flip_status (void)
Parameters: None
Returns: either KEY_FLIP_OPEN (12) or KEY_FLIP_CLOSED (13) according to the flip status.
Functional description: Reads the GPIO associated with the flip and returns its current status (either open or closed).

5.6.2.4 *KEY_power_down_allowed*

Prototype: int KEY_power_down_allowed (void)
Parameters: None.
Returns: TRUE if power down is allowed.
Functional description: Returns the key_power_down_allowed_flag, which is reset when debouncing is ongoing and set when it’s completed.

5.6.2.5 *KEY_confirm_sdl_mail*

Prototype: void KEY_confirm_sdl_mail (void)

Parameters: None.
Functional description: Decrements key_mail_confirm_counter, which is used by the MMI to confirm the reception of the keyboard SDL signal in order to avoid the overflow of the MMI queue. If key_mail_confirm_counter reaches its maximum value then any further key press and release actions are ignored.

5.6.2.6 *KEY_get_key_status*

Prototype: ubyte KEY_get_key_status (int key_value)
Parameters: key_value is the value of a specific key whose status must be tested.
Returns: TRUE If the key specified by key_value is pressed, FALSE otherwise.
Functional description: Checks whether the specified key is pressed or not. If more keys have the same key_value, only the key with the lowest index in the KEY_MAP array is checked. The function is only available when multi key support is enabled.

5.6.2.7 *KEY_get_all_keys_status*

Prototype: char *KEY_get_all_keys_status (void)
Parameters: None.
Returns: a char pointer to the key status array
Functional description: Returns the current status of the keyboard matrix via a global status array, updated after each successful debounce.

5.6.2.8 *KEY_get_multikey_status*

Prototype: ubyte KEY_get_multikey_status (void)
Parameters: None.
Returns: TRUE if multiple key detection mode is supported, FALSE otherwise
Functional description: Multiple keys detection is supported by defining KEY_MULTI.

5.6.2.9 *KEY_set_multikey_mode*

Prototype: void KEY_set_multikey_mode (char mode)
Parameters: mode enables (TRUE) and disables (FALSE) the multiple key detection mode.
Functional description: Enables multiple key detection mode run-time (only if.multi mode is supported by defining KEY_MULTI).

5.6.3 *Production test interface*

5.6.3.1 *KEY_test_reset_key_matrix*

Prototype: void KEY_test_reset_key_matrix (void)
Parameters: None.
Functional description: Resets KEY_matrix, used for production tests.

5.6.3.2 *KEY_test_get_key_matrix*

Prototype: key_matrix_type KEY_test_get_key_matrix (void)
Parameters: None.
Returns: KEY_matrix, used for production tests.
Functional description: KEY_matrix is updated on each key press and key release.

5.6.3.3 *KEY_test_hash_star_key_pressed*

Prototype: unsigned int KEY_test_hash_star_key_pressed (void)
Parameters: None.
Returns: TRUE if both hash and star keys are simultaneously pressed, FALSE otherwise.
Functional description: Initialises the keyboard driver then checks the star and hash keys status.

5.6.4 *Simulation interface*

5.6.4.1 *KEY_ext_key_press_simulation*

Prototype: void KEY_ext_key_press_simulation (int key_value)
Parameters: key_value is the key whose press simulation is required.
Functional description: Sets a simulation flag, stores the required key value then forces a KPDIC interrupt, which triggers an SDL Signal to the MMI.

5.6.4.2 *KEY_ext_multi_key_press_simulation*

Prototype: void KEY_ext_multi_key_press_simulation (int keyarray[3])
Parameters: keyarray contains the value of the keys whose press simulation is required.
Functional description: Sets a simulation flag, stores the required key values then forces a KPDIC interrupt, which triggers the corresponding SDL Signal to the MMI. This function is active only if multiple keys simulation is supported by defining KEY_MULTI_SIM.

5.7 Keyboard driver configuration

The keyboard driver can be configured by means of both defines and flags, enabling or disabling the following features.

- Multiple keys detection: is supported if KEY_MULTI is defined and is enabled if key_multikey_mode_flag is set (by calling KEY_set_multikey_mode).
- MMI queue protection: is supported if KEY_MAIL_CONFIRM is defined, sets a limit to the number of SDL Signals that the keyboard driver sends to the MMI without receiving any confirmation. If such limit is exceeded, the keyboard driver discards any further key press and release.
- Early initialization: anticipates the keyboard driver start up, is supported if KEY_EARLY_INIT is defined.
- New scan algorithm: avoids short circuits among output pins in multi key mode, is supported if KEY_NEW_SCAN is defined.
- Key flip: notifies the MMI each time the flip is opened or closed, is supported if KEY_FLIP is defined.

6 References

6.1 External

E-GOLDlite GSM/GPRS Baseband System manual, PMB 7860 Design Specification, Rev. 1.12, 2003-11-25 (Infineon Technologies AG)

6.2 Internal

None.

7 Document change report

	Change Reference		Record of changes made to previous released version	
Rev	**Date**	**CR**	**Section**	**Comment**
1.0	12/12/2003	N.A.	Document created	
			§ 5.1, § 5.4, Table 4-1 and 4.2	Upgrade to Globe 2, added hook key handling
2.0	15/01/2006		Review all document	Upgrade to Globe6, Goldfinch

8 Approval

Revision	Approver(s)	Date	Source/signature
1.0	Stefano Godeas	12/12/2003	Document stored on server
1.1	Stefano Godeas	24/05/2004	Document stored on server
2.0	Stefano Godeas	10/02/2006	Document stored on server

9 Annex 1: new keypad scan algorithm

In order to fully support multiple keys detection a new keypad scan algorithm has been implemented. The purpose of such algorithm is to avoid short circuits that occur in the standard scanning procedure when 2 or more keys on the same row are simultaneously pressed. Infact suppose that both PB11 and PB12 are pressed: this causes a short circuit between KP1 and KP2 when KP1 is reset during the keyboard matrix scan, as shown in Figure 9-1.

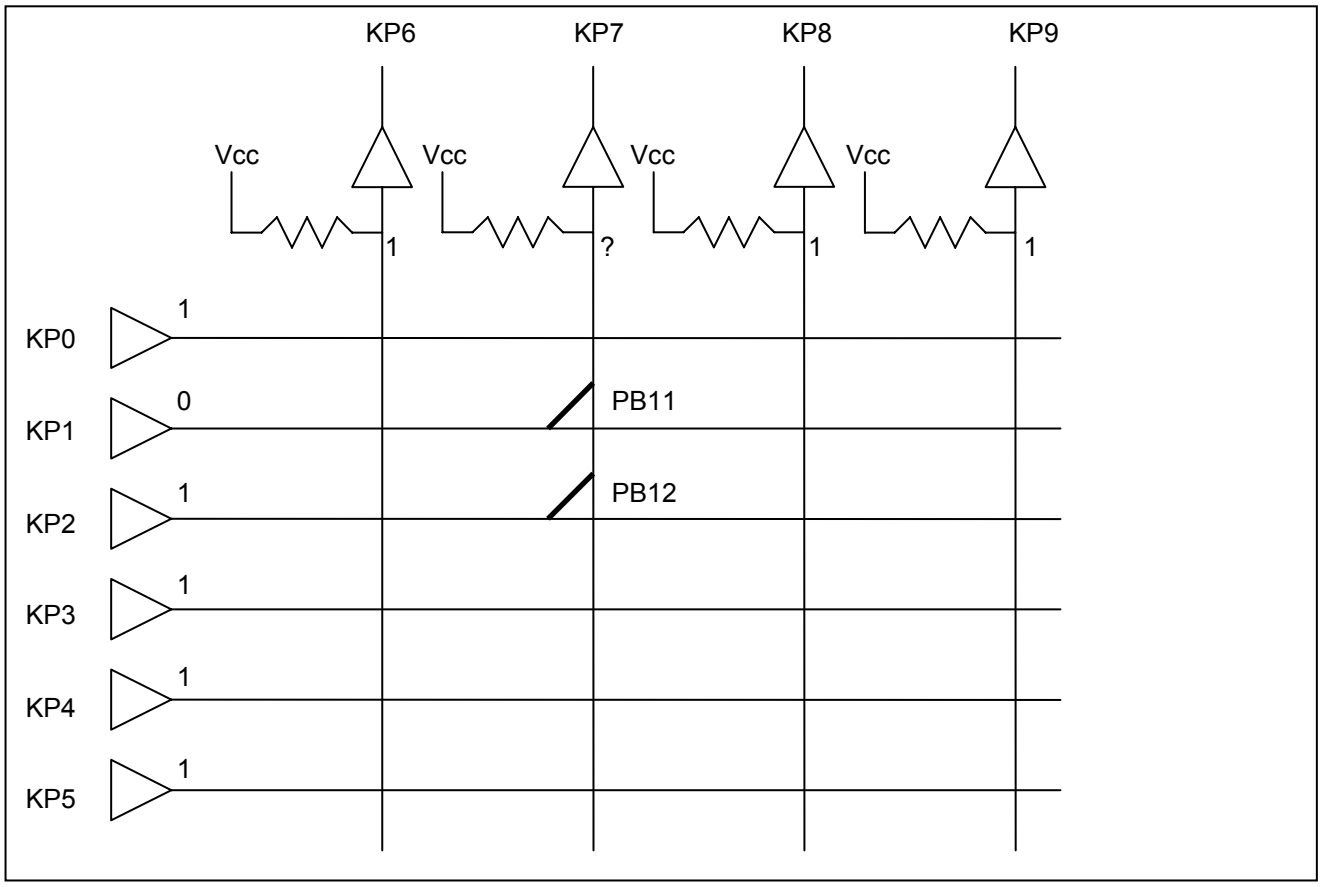

Figure 9-1

The short circuit is due to the KP0…KP5 configuration that does not use any pull-up resistor, as shown in Figure 9-2.

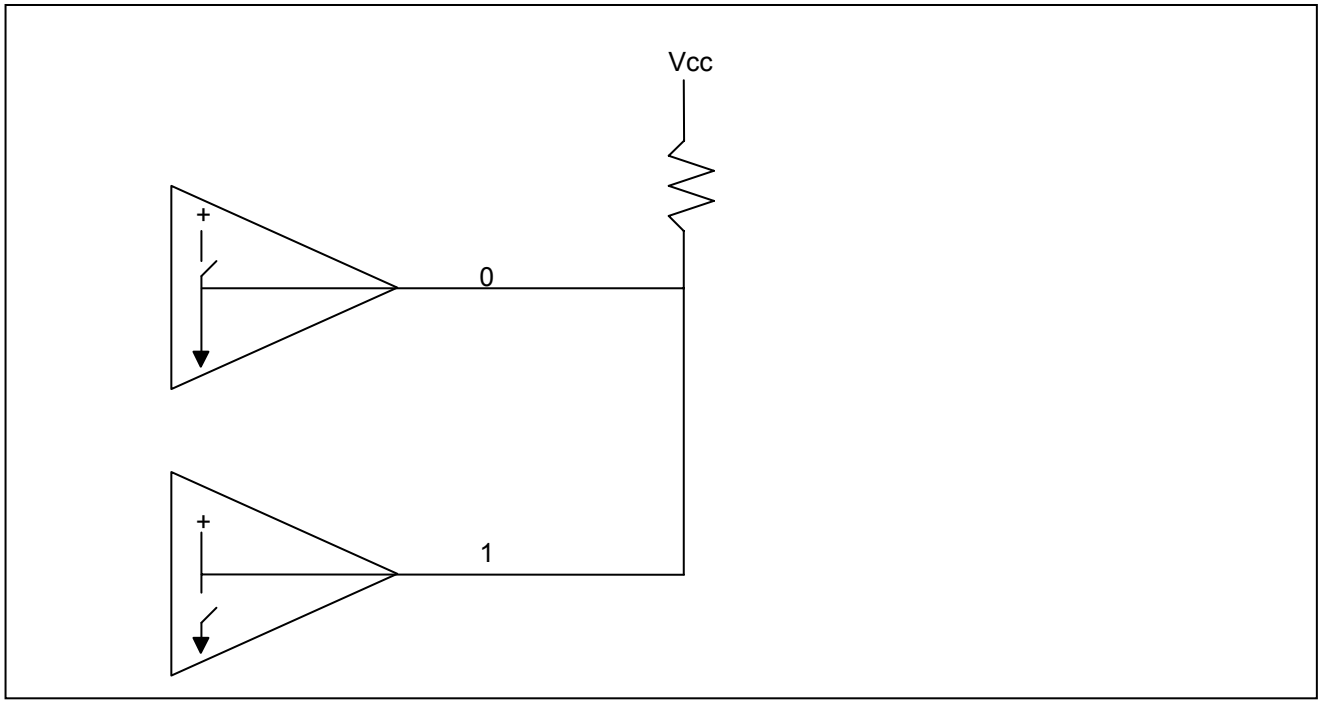

Figure 9-2

In order to avoid this situation KP0…KP5 are configured as general purposes GPIOs with pull-up resistors; once a keypad interrupt fires the scan procedure is modified as follows:

Initialization:

- KP0…KP5 are first configured as GPIO open drain output pins with pull-up resistors, avoiding any possible short circuit;
- KP0…KP5 are then configured as GPIO push-pull output pins to speed up the output signal transition;
- KP0…KP5 are configured again as GPIO open drain output pins with pull-up resistors (see Figure 9-3)

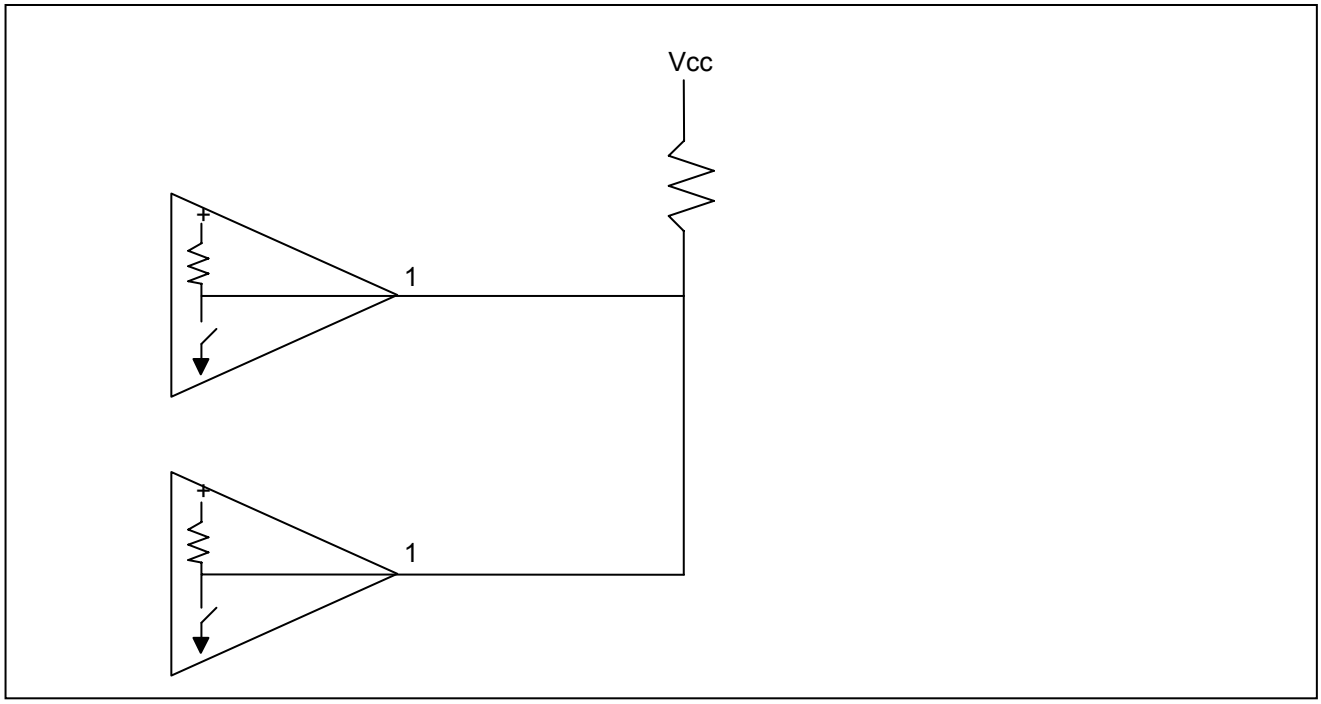

Figure 9-3

Scanning (for n=0,1,…5):

- KPn is configured as GPIO push-pull output pin and is reset; KBDINP is then read bit per bit detecting the status of all the keys in column n. As the other output pins are in open drain with pull-up configuration and are set, no short circuit is possible even though other keys are simultaneously pressed (see Figure 9-4, representing the same situation of Figure 9-2 i.e. both PB11 and PB12 are pressed);
- KPn is then configured as GPIO open drain with pull-up and is set, avoiding any short circuit (back as in Figure 9-3);
- Now in this safe situation KPn is configured as GPIO push-pull, to speed up the signal transition from 0 to 1 (see Figure 9-5);
- KPn is configured again as GPIO open drain with pull-up, to be ready for scanning the next column (see Figure 9-3 again);

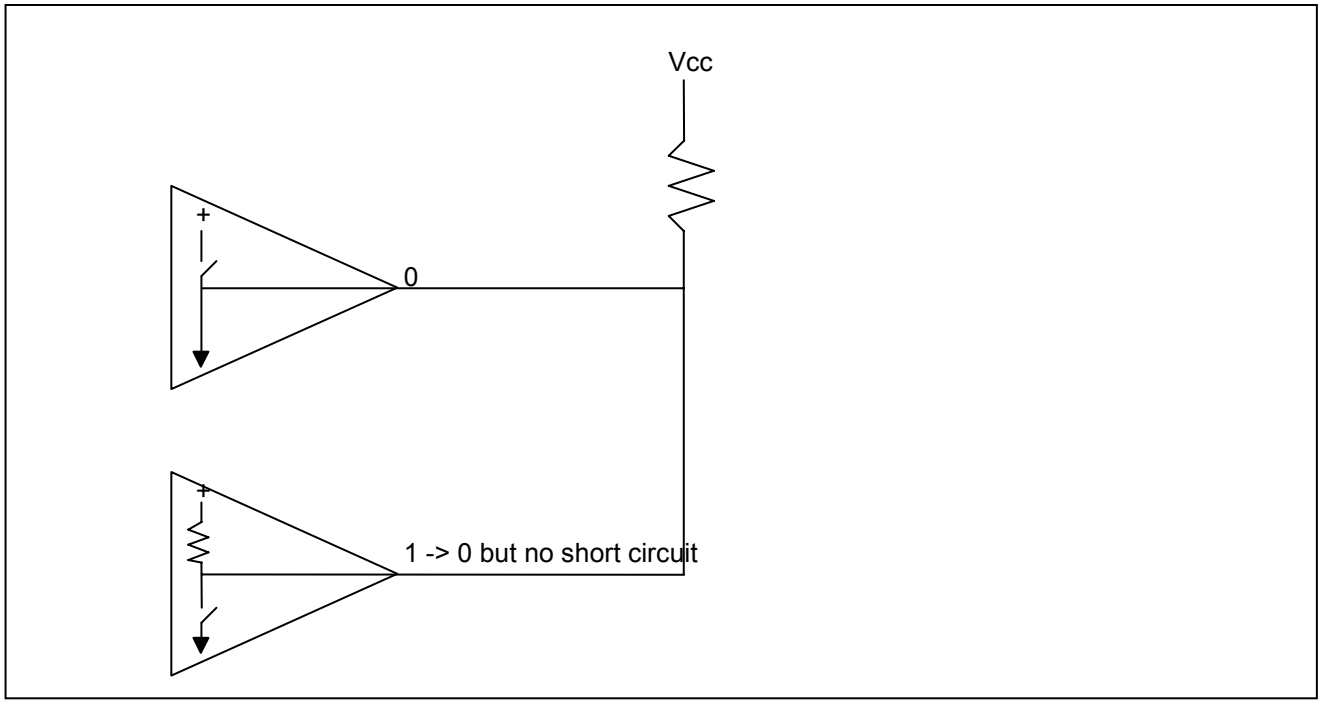

Figure 9-4

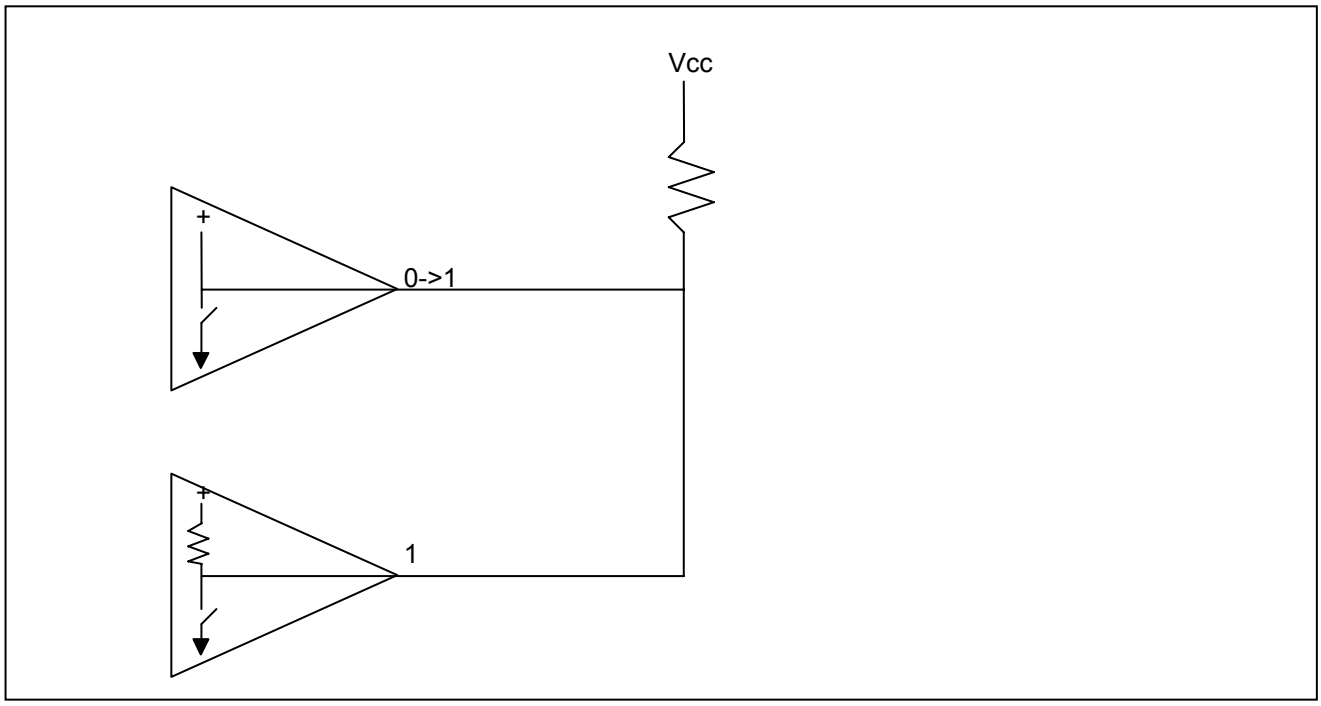

Figure 9-5

Howether the keyboard driver is not able to support all possible combination of simultaneous key press; this is not related to the driver implementation but to the hardware design of the keyboard as a matrix and can't be solved by Software. In particular when simultaneously pressing three keys placed in three corners of a rectangle, also the key placed in the fourth corner will be erroneously detected as pressed and it is impossible to detect which one it is; this is due to the short circuit created by such combination of key press, and it's unavoidable. All other combinations are allowed, for example the driver correctly detects:

- All keys in the same column, for ex. *, 0, # and UP (see table 5-1);
- All keys in the same row, for ex. LSO, SND, 1, 4, 7, *;
- In general any combination of keys not displaced in at least 3 corners of a rectangle, for ex. *, 2, 6, LEFT, OK or LSO, RSO, LEFT, DOWN, OK etc.

If 1, 3 and 7 are pressed then also 9 is erroneously detected as if being pressed, therefore these combinations will not be supported by the keyboard drivers.

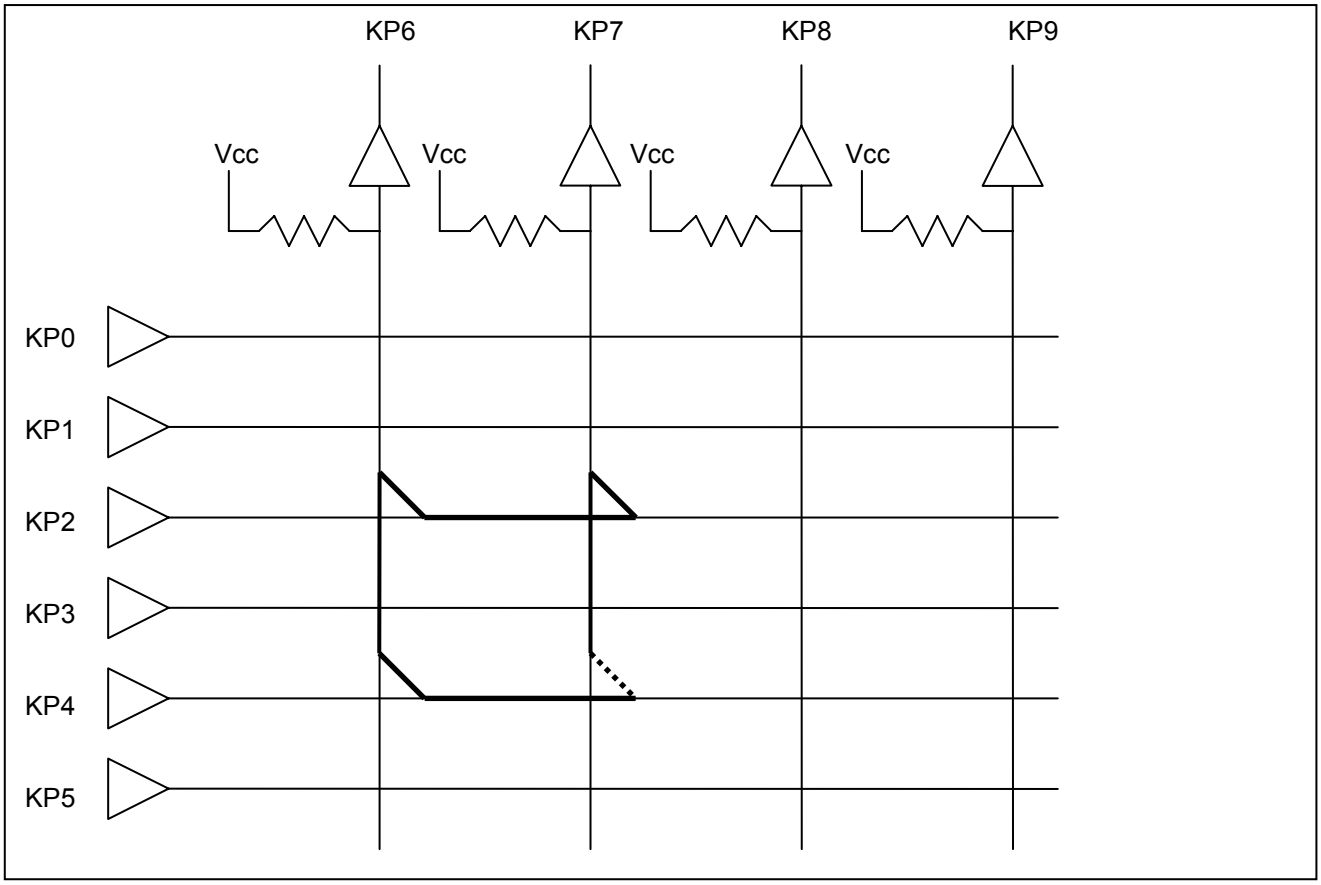

Figure 9-6

10 Annex2

None.

11 Annex 3

None.