

Game Input Devices -- User Manual

Introduction

PicoMite MMBasic supports several input methods suitable for game development. This manual covers all available game input devices -- from simple keyboard and GPIO buttons to USB gamepads and Wii controllers. Each section explains the commands, functions, and practical patterns needed to integrate that input method into a game.

Input Methods at a Glance

Method	Hardware Needed	Build Requiremen	Analog Axes	Buttons	Best For
Keyboard (INKEY\$)	Any keyboard (USB/PS2/Serial)	Any build	No	Yes	Text input, simple controls
Keyboard (KEYDOWN)	USB or PS2 keyboard	Any build	No	Yes (6 simultaneous)	Multi-key game controls
USB Gamepad	USB gamepad (PS4/PS3/Xbox/Generic)	USB builds only	Up to 6 axes	Up to 16	Full-featured game input
Wii Nunchuck	Wii Nunchuck via I2C	Any build	Joystick + accelerometer	2 (C, Z)	Tilt-based games
Wii Classic	Wii Classic Controller via I2C	Any build	2 sticks + triggers	15	Console-style games
GPIO Buttons	Physical buttons + wiring	Any build	Yes (on analog-capable pins)	Limited by pins	Arcade cabinets, custom builds
PS/2 Raw Scancodes	PS/2 keyboard	Non-USB builds	No	Yes	Low-level key detection
Touch Screen	SPI resistive or I2C capacitive	Any build with LCD	X/Y coordinates	Touch down/up	Touch-based games, menus

USB builds are: PicoMiteUSB, PicoMiteVGAUSB, and PicoMiteHDMIUSB.

Keyboard Input

Keyboard input is available on all PicoMite builds. The KEYDOWN function supports both USB and PS2 keyboards, providing simultaneous multi-key detection for game development. For game development, non-blocking techniques are essential -- you need to check for key presses without halting the game loop.

INKEY\$ -- Non-Blocking Character Input

INKEY\$ returns the next character from the console input buffer, or an empty string if no key has been pressed. It never blocks.

Syntax:

```
key$ = INKEY$
```

Returns:

- Empty string "" -- no key pressed
- Single character -- the ASCII character of the key pressed

Game Input Devices -- User Manual

Special Keys: MMBasic automatically converts VT100 escape sequences from the keyboard into single-character codes in the range 128-156. INKEY\$ returns these as a single character -- no multi-byte parsing is needed.

Key	Code	CHR\$
Up Arrow	128	CHR\$(128)
Down Arrow	129	CHR\$(129)
Left Arrow	130	CHR\$(130)
Right Arrow	131	CHR\$(131)
Insert	132	CHR\$(132)
Home	134	CHR\$(134)
End	135	CHR\$(135)
Page Up	136	CHR\$(136)
Page Down	137	CHR\$(137)
Delete	127	CHR\$(127)
F1-F12	145-156	CHR\$(145)-CHR\$(156)

Example -- Arrow Key Movement with INKEY\$:

```
DO
  k$ = INKEY$
  IF k$ = CHR$(128) THEN y = y - 1    ' Up arrow
  IF k$ = CHR$(129) THEN y = y + 1    ' Down arrow
  IF k$ = CHR$(130) THEN x = x - 1    ' Left arrow
  IF k$ = CHR$(131) THEN x = x + 1    ' Right arrow
  IF k$ = " " THEN fire = 1           ' Space bar
  ' ... render frame ...
LOOP
```

Limitations:

- Only detects one key per call (no simultaneous key presses)
- Buffer can accumulate characters if not read frequently

KEYDOWN -- Multi-Key Detection (USB and PS2)

KEYDOWN provides the most powerful keyboard input for games. It detects up to 6 simultaneous key presses plus modifier keys. On USB builds it reads the USB HID key report directly; on non-USB builds it tracks PS2 scancode make/break events.

Requirement: Any build with a USB or PS2 keyboard connected.

Syntax:

```
value = KEYDOWN(n)
```

n	Returns	Description
0	0-6	Number of keys currently pressed
1	Key code	First key currently pressed (0 if none)
2	Key code	Second key currently pressed (0 if none)
3	Key code	Third key currently pressed (0 if none)
4	Key code	Fourth key currently pressed (0 if none)
5	Key code	Fifth key currently pressed (0 if none)
6	Key code	Sixth key currently pressed (0 if none)
7	Modifier bitmask	Modifier keys currently held (see below)
8	Lock state	Caps/Num/Scroll lock states

Game Input Devices -- User Manual

Note: Each call to KEYDOWN clears the console receive buffer, preventing character accumulation.

Modifier Bitmask (KEYDOWN(7)):

The modifier bitmask reports which modifier keys are currently held. USB and PS2 keyboards report different modifiers due to hardware differences.

USB Keyboard Modifiers:

Bit	Value	Modifier
0	1	Left Ctrl
1	2	Left Shift
2	4	Left Alt
3	8	Left GUI (Windows key)
4	16	Right Ctrl
5	32	Right Shift
6	64	Right Alt
7	128	Right GUI

PS2 Keyboard Modifiers:

Bit	Value	Modifier
0	1	Left Alt
1	2	Ctrl
3	8	Left Shift
4	16	Right Alt (AltGr)
7	128	Right Shift

Note: PS2 keyboards do not report GUI (Windows) keys, and left/right Ctrl are not distinguished. Bits 2, 5, and 6 are unused on PS2.

Lock State (KEYDOWN(8)):

Bit	Value	Lock
0	1	Caps Lock
1	2	Num Lock
2	4	Scroll Lock (USB only)

Key Code Reference:

Code	Key	Code	Key	Code	Key
27	ESC	128	Up Arrow	145	F1
13	Enter	129	Down Arrow	146	F2
32	Space	130	Left Arrow	147	F3
9	Tab	131	Right Arrow	148	F4
8	Backspace	132	Insert	149	F5
127	Delete	134	Home	150	F6
139	Alt	135	End	151	F7
140	Shift	136	Page Up	152	F8
141	Ctrl	137	Page Down	153	F9
154	F10				
48-57	0-9	97-122	a-z	155	F11
65-90	A-Z	156	F12		

Game Input Devices -- User Manual

Example -- Multi-Key Game Input:

```
DO
  n = KEYDOWN(0)          ' number of keys held
  up = 0 : dn = 0 : lf = 0 : rt = 0 : fire = 0
  FOR i = 1 TO n
    k = KEYDOWN(i)
    IF k = 128 THEN up = 1  ' Up arrow
    IF k = 129 THEN dn = 1  ' Down arrow
    IF k = 130 THEN lf = 1  ' Left arrow
    IF k = 131 THEN rt = 1  ' Right arrow
    IF k = 32 THEN fire = 1 ' Space
  NEXT i
  ' Diagonal movement works naturally - both up+left detected simultaneously
  x = x + rt - lf
  y = y + dn - up
  ' ... render frame ...
LOOP
```

Example -- WASD with Shift for Run:

```
DO
  n = KEYDOWN(0)
  mods = KEYDOWN(7)
  speed = 1
  IF mods AND 2 THEN speed = 3  ' Left Shift held = run
  dx = 0 : dy = 0
  FOR i = 1 TO n
    k = KEYDOWN(i)
    SELECT CASE k
      CASE ASC("w"), ASC("W") : dy = -speed
      CASE ASC("s"), ASC("S") : dy = speed
      CASE ASC("a"), ASC("A") : dx = -speed
      CASE ASC("d"), ASC("D") : dx = speed
    END SELECT
  NEXT i
  x = x + dx : y = y + dy
  ' ... render frame ...
LOOP
```

ON KEY -- Interrupt-Driven Key Input

ON KEY provides interrupt-driven keyboard input. When a key arrives in the console buffer, BASIC jumps to a handler subroutine.

Syntax:

```
ON KEY handler          ' any key triggers the interrupt
ON KEY keycode, handler ' only the specified key triggers it
ON KEY 0                 ' disable
```

- handler -- a SUB or label to jump to
- keycode -- ASCII value of the specific character to trigger on

Example -- Pause on Escape:

```
ON KEY 27, PauseGame
' ... game loop ...
```

Game Input Devices -- User Manual

```
SUB PauseGame
  PRINT "PAUSED - press any key"
  k$ = INPUT$(1, #0)
END SUB
```

Note: ON KEY responds to console/keyboard input only, not GPIO pins. It has higher interrupt priority than pin interrupts and tick timers.

ON PS2 -- Raw PS/2 Scancode Interrupts

ON PS2 provides interrupt-driven access to raw PS/2 scancodes before MMBasic's keyboard driver processes them. This gives you low-level key-down and key-up events for every key, enabling simultaneous key detection on non-USB builds.

Requirement: Non-USB builds only (PicoMite, PicoMiteVGA, PicoMiteHDMI) with a PS/2 keyboard configured via OPTION KEYBOARD.

Syntax:

```
ON PS2 handler      ' enable raw scancode interrupt
ON PS2 0             ' disable
```

When a PS/2 scancode arrives, MMBasic calls handler and makes the raw scancode available via MM.PS2 (or equivalently MM.INFO(PS2)).

MM.PS2 / MM.INFO(PS2) -- Reading the Raw Scancode

Syntax:

```
code% = MM.PS2
code% = MM.INFO(PS2)      ' equivalent
```

Both return the same value -- the raw PS/2 scancode with flags:

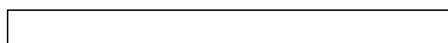
Value Pattern	Meaning
&H00xx	Key down, scancode xx
&HF0xx	Key up, scancode xx
&HE0xx	Extended key down (e.g. arrow keys, Home, End)
&HE0F0xx	Extended key up
0	No event / cleared

Common PS/2 Scancodes (Set 2):

Key	Down Code	Extended?
A	&H1C	No
W	&H1D	No
S	&H1B	No
D	&H23	No
Space	&H29	No
Enter	&H5A	No
Left Shift	&H12	No
Up Arrow	&HE075	Yes
Down Arrow	&HE072	Yes

Game Input Devices -- User Manual

Left Arrow



Game Input Devices -- User Manual

&HE06B

Game Input Devices -- User Manual

Yes

Game Input Devices -- User Manual

Right Arrow	&HE074	Yes
-------------	--------	-----

Example -- Tracking Key-Down/Key-Up State:

```
DIM key_w, key_a, key_s, key_d, key_space

ON PS2 HandlePS2

' Game loop
DO
  dx = 0 : dy = 0
  IF key_a THEN dx = dx - 1
  IF key_d THEN dx = dx + 1
  IF key_w THEN dy = dy - 1
  IF key_s THEN dy = dy + 1
  IF key_space THEN fire = 1
  x = x + dx * speed
  y = y + dy * speed
  ' ... render frame ...
LOOP

SUB HandlePS2
  LOCAL sc% = MM.PS2
  LOCAL down = NOT (sc% AND &HF000) ' 0 if key-up
  LOCAL code% = sc% AND &HFF
  SELECT CASE code%
    CASE &H1D : key_w = down
    CASE &H1C : key_a = down
    CASE &H1B : key_s = down
    CASE &H23 : key_d = down
    CASE &H29 : key_space = down
  END SELECT
END SUB
```

This technique gives simultaneous key detection on non-USB builds -- something INKEY\$ alone cannot provide. ON PS2 is also useful when you need the raw scancode values rather than the ASCII-mapped characters returned by KEYDOWN.

Note: ON PS2 and MM.PS2 are not available on USB keyboard builds (PicoMiteUSB etc.). Those builds use KEYDOWN instead. On non-USB builds, both KEYDOWN and ON PS2 are available -- KEYDOWN is simpler (ASCII character codes, auto-compacted array), while ON PS2 gives raw scancode-level control.

PS2 KEYDOWN vs ON PS2 -- When to Use Which

Feature	KEYDOWN (PS2)	ON PS2
Key reporting	ASCII character codes	Raw PS2 scancodes
Simultaneous keys	Up to 6	Unlimited (manual tracking)
Modifier keys	Bitmask in slot 7	Manual flag tracking
Key rollover	Limited by PS2 keyboard hardware (typically 2-3 keys, some NKRO)	Same hardware limit
Array management	Auto-compacted, left-justified	Manual
Interrupt-driven	No (polled)	Yes
Best for	Simple multi-key game input	Low-level key event processing

Game Input Devices -- User Manual

USB Gamepads

USB gamepads provide the richest input for game development -- analog sticks, triggers, d-pad, face buttons, and on PS4 controllers, motion sensors, haptic feedback, and a colour light bar.

Requirement: USB builds only (PicoMiteUSB, PicoMiteVGAUSB, PicoMiteHDMIUSB).

Supported Controllers

Type	Type	Controllers
PS4 (DualShock 4)	128	Sony DualShock 4, Hori FC4, Hori PS4 Mini
PS3 (DualShock 3)	129	Sony DualShock 3
Generic (SNES-style)	130	Various USB gamepads (7 pre-configured, more via CONFIGURE)
Xbox-style	131	EasySMX and similar

USB HID Channels

The system has 4 HID slots:

- Channel 1: Reserved for keyboard (if connected)
- Channel 2: Reserved for mouse (if connected)
- Channels 3-4: Preferred for gamepads

If keyboard/mouse are not connected, their slots can be used by gamepads. Up to 4 USB HID devices total.

Reading Gamepad State: DEVICE()

The DEVICE() function reads all gamepad data.

Syntax:

```
value = DEVICE(GAMEPAD n, parameter$)
```

Where n is the channel number (1-4) and parameter\$ is one of:

Parameter	Returns	Range	Description
"LX"	Integer	0-255	Left stick X axis (128 = centre)
"LY"	Integer	0-255	Left stick Y axis (128 = centre)
"RX"	Integer	0-255	Right stick X axis (128 = centre)
"RY"	Integer	0-255	Right stick Y axis (128 = centre)
"L"	Integer	0-255	Left analog trigger pressure
"R"	Integer	0-255	Right analog trigger pressure
"B"	Integer	0-65535	Button bitmap (16-bit)
"T"	Integer	128-131	Controller type code
"GX"	Integer	int16	Gyroscope X (PS4 only)
"GY"	Integer	int16	Gyroscope Y (PS4 only)
"GZ"	Integer	int16	Gyroscope Z (PS4 only)
"AX"	Integer	int16	Accelerometer X (PS4 only)
"AY"	Integer	int16	Accelerometer Y (PS4 only)
"AZ"	Integer	int16	Accelerometer Z (PS4 only)
"RAW"	String	bytes	Raw HID report data

Notes:

Game Input Devices -- User Manual

- Analog axes (LX, LY, RX, RY, L, R) are only populated for PS4, PS3, and Xbox controllers
- Generic/SNES gamepads only provide "B" (buttons) and "T" (type)
- Gyroscope and accelerometer data are PS4 only

Button Bitmap

The button bitmap returned by DEVICE(GAMEPAD n, "B") has these bit positions:

Bit	Value	Button	PS4	Xbox
0	1	R1 / RB	R1	RB
1	2	Start	Options	Start
2	4	Home	PS button	Xbox
3	8	Select	Share	Back
4	16	L1 / LB	L1	LB
5	32	D-pad Down	Down	Down
6	64	D-pad Right	Right	Right
7	128	D-pad Up	Up	Up
8	256	D-pad Left	Left	Left
9	512	R2 (digital)	R2	RT
10	1024	Triangle / Y	Triangle	Y
11	2048	Circle / B	Circle	B
12	4096	Square / X	Square	X
13	8192	Cross / A	Cross	A
14	16384	L2 (digital)	L2	LT
15	32768	Touchpad	Touchpad	N/A

PS4 d-pad uses a hat field internally -- diagonal directions (e.g., Up+Right) set both corresponding bits.

Example -- Reading Buttons and Sticks:

```
ch = 3      ' gamepad on channel 3
DO
  buttons = DEVICE(GAMEPAD ch, "B")
  lx = DEVICE(GAMEPAD ch, "LX")
  ly = DEVICE(GAMEPAD ch, "LY")

  ' Check d-pad
  IF buttons AND 128 THEN dy = -1  ' Up
  IF buttons AND 32 THEN dy = 1   ' Down
  IF buttons AND 256 THEN dx = -1 ' Left
  IF buttons AND 64 THEN dx = 1   ' Right

  ' Use left stick with deadzone
  IF ABS(lx - 128) > 20 THEN
    x = x + (lx - 128) / 32
  END IF
  IF ABS(ly - 128) > 20 THEN
    y = y + (ly - 128) / 32
  END IF

  ' Face buttons
  IF buttons AND 8192 THEN fire = 1  ' Cross / A
  IF buttons AND 4096 THEN jump = 1  ' Square / X
  ' ... render frame ...
```

Game Input Devices -- User Manual

LOOP

Gamepad Interrupts

Instead of polling every frame, you can set up an interrupt that fires when button states change.

Enable:

```
GAMEPAD INTERRUPT ENABLE n, handler [, mask]
```

- n -- channel number (1-4)
- handler -- SUB or label to call
- mask -- (optional) 16-bit bitmask (default 65535 = all buttons). Only button changes matching the mask trigger the interrupt.

The interrupt fires when (new_buttons XOR old_buttons) AND mask is non-zero.

Disable:

```
GAMEPAD INTERRUPT DISABLE n
```

Example -- Interrupt on D-pad Changes:

```
' Only trigger on d-pad buttons (bits 5-8 = value 480)
GAMEPAD INTERRUPT ENABLE 3, DpadHandler, 480
```

```
SUB DpadHandler
  LOCAL b = DEVICE(GAMEPAD 3, "B")
  IF b AND 128 THEN PRINT "Up"
  IF b AND 32 THEN PRINT "Down"
  IF b AND 256 THEN PRINT "Left"
  IF b AND 64 THEN PRINT "Right"
END SUB
```

PS4-Specific Features

Haptic Feedback (Rumble)

```
GAMEPAD HAPTIC n, left_motor, right_motor
```

- n -- channel (1-4, PS4 only)
- left_motor -- heavy motor intensity, 0-255
- right_motor -- light motor intensity, 0-255

Both motors run continuously at the specified intensity; set to 0 to stop.

```
GAMEPAD HAPTIC 3, 200, 100    ' strong left, medium right
PAUSE 500
GAMEPAD HAPTIC 3, 0, 0        ' stop
```

Light Bar Colour

```
GAMEPAD COLOUR n, colour
```

- n -- channel (1-4, PS4 only)
- colour -- 24-bit RGB value

Game Input Devices -- User Manual

```
GAMEPAD COLOUR 3, RGB(255, 0, 0)    ' red light bar
GAMEPAD COLOUR 3, RGB(0, 255, 0)    ' green
GAMEPAD COLOUR 3, RGB(0, 0, 0)      ' off
```

Motion Sensors

PS4 controllers have a 3-axis gyroscope and accelerometer:

```
gx = DEVICE(GAMEPAD 3, "GX")  ' gyroscope X (int16)
gy = DEVICE(GAMEPAD 3, "GY")  ' gyroscope Y
gz = DEVICE(GAMEPAD 3, "GZ")  ' gyroscope Z
ax = DEVICE(GAMEPAD 3, "AX")  ' accelerometer X (int16)
ay = DEVICE(GAMEPAD 3, "AY")  ' accelerometer Y
az = DEVICE(GAMEPAD 3, "AZ")  ' accelerometer Z
```

GAMEPAD MONITOR -- Identifying Unknown Controllers

If you have a USB gamepad that is not automatically recognised, use GAMEPAD MONITOR to discover its button mappings.

```
GAMEPAD MONITOR
```

When active, the system prints the raw HID report bytes when any button is pressed. Compare the hex bytes before and after pressing each button to identify which byte index and bit correspond to which button.

Use GAMEPAD MONITOR SILENT to enter monitor mode without printing the raw data (useful for accepting unrecognised controllers without output).

GAMEPAD CONFIGURE -- Adding New Controllers

Once you have identified the byte offsets and bit codes using MONITOR mode, you can configure an unsupported controller:

```
GAMEPAD CONFIGURE vid, pid, R_idx, R_code, START_idx, START_code, HOME_idx, HOME_code, SELECT_idx,
SELECT_code, L_idx, L_code, DOWN_idx, DOWN_code, RIGHT_idx, RIGHT_code, UP_idx, UP_code, LEFT_idx,
LEFT_code, R2_idx, R2_code, X_idx, X_code, A_idx, A_code, Y_idx, Y_code, B_idx, B_code, L2_idx,
L2_code, TOUCH_idx, TOUCH_code
```

34 parameters:

- vid, pid -- USB Vendor ID and Product ID (from MONITOR output)
- Then 16 button pairs (byte_index, code) in this order: R1, Start, Home, Select, L1, Down, Right, Up, Left, R2, X, A, Y, B, L2, Touch

Code values:

- 0-7 -- button is set if bit N of the report byte is set
- 128-135 -- button is set if bit (code-128) is NOT set (inverted)
- 64 -- button is set if report byte < 64 (axis low)
- 192 -- button is set if report byte > 192 (axis high)
- Set index to 255 if the button does not exist on the controller

Pre-configured Generic Gamepads:

VID	PID	Description
0x0810	0xE501	EasySMX / generic

Game Input Devices -- User Manual

0x0079	0x0011	Generic USB gamepad
0x081F	0xE401	Generic USB gamepad
0x1C59	0x0026	GameSir
0x06A3	0x0107	Saitek
0x11FF	0x3331	DragonRise / generic
0x0583	0x2060	Padix / generic

Wii Controllers

PicoMite supports two Wii controllers over I2C: the Nunchuck and the Classic Controller. These work on all PicoMite builds (no USB requirement) and connect via the SYSTEM I2C bus.

Prerequisites

Both controllers communicate over I2C at address 0x52. You must configure SYSTEM I2C before opening either controller:

```
SYSTEM I2C sda_pin, scl_pin, speed
```

Only one Wii controller (Nunchuck or Classic) can be open at a time.

Wii Nunchuck

The Nunchuck provides a joystick (X/Y), 3-axis accelerometer, and two buttons (C and Z).

Opening and Closing

```
WII NUNCHUCK OPEN [interrupt_handler]
WII NUNCHUCK CLOSE
```

- interrupt_handler -- (optional) SUB or label called when C or Z is pressed (rising edge only)
- The command initialises I2C communication, verifies the device ID, and begins polling at ~10ms intervals

Errors:

- "SYSTEM I2C not configured" -- call SYSTEM I2C first
- "Nunchuck not connected" -- I2C initialisation failed
- "Device connected is not a Nunchuck" -- wrong device ID
- "Nunchuck not responding" -- no valid data received

Reading Nunchuck Data: DEVICE()

```
value = DEVICE(NUNCHUCK parameter)
```

Parameter	Returns	Range	Description
JX	Integer	0-255	Joystick X (centre ~128)
JY	Integer	0-255	Joystick Y (centre ~128)
AX	Integer	0-1023	Accelerometer X (10-bit)
AY	Integer	0-1023	Accelerometer Y (10-bit)
AZ	Integer	0-1023	Accelerometer Z (10-bit)

Game Input Devices -- User Manual

Z	Integer	0 or 1	Z button (1 = pressed)
C	Integer	0 or 1	C button (1 = pressed)
T	Integer	Device type ID	

Calibration values are also available:

Parameter	Description
AX0, AY0, AZ0	Accelerometer zero-G calibration
AX1, AY1, AZ1	Accelerometer one-G calibration
JXL, JXC, JXR	Joystick X: left limit, centre, right limit
JYT, JYC, JYB	Joystick Y: top limit, centre, bottom limit

Example -- Nunchuck Tilt Control:

```
SYSTEM I2C GP4, GP5, 100000
WII NUNCHUCK OPEN

DO
  jx = DEVICE(NUNCHUCK JX)
  jy = DEVICE(NUNCHUCK JY)
  z_btn = DEVICE(NUNCHUCK Z)
  c_btn = DEVICE(NUNCHUCK C)

  ' Apply deadzone to joystick
  IF ABS(jx - 128) > 10 THEN
    player_x = player_x + (jx - 128) / 32
  END IF
  IF ABS(jy - 128) > 10 THEN
    player_y = player_y - (jy - 128) / 32
  END IF

  IF z_btn THEN fire = 1
  IF c_btn THEN jump = 1
  ' ... render frame ...
LOOP

WII NUNCHUCK CLOSE
```

Nunchuck Interrupts

When an interrupt handler is specified with WII NUNCHUCK OPEN handler, it fires on rising edges of the C or Z buttons (transition from not-pressed to pressed). Inside the handler, read DEVICE(NUNCHUCK C) and DEVICE(NUNCHUCK Z) to determine which button triggered the interrupt.

Wii Classic Controller

The Classic Controller provides two analog sticks, two analog triggers, a d-pad, and 15 buttons -- similar to a console gamepad.

Opening and Closing

```
WII CLASSIC OPEN interrupt_handler [, button_mask]
WII CLASSIC CLOSE
```

Game Input Devices -- User Manual

Also accepted as:

```
WII OPEN interrupt_handler [, button_mask]
WII CLOSE
```

- interrupt_handler -- SUB or label called when specified buttons change state
- button_mask -- (optional) 15-bit mask (0-32767) selecting which buttons trigger the interrupt. Default: all buttons.

Errors:

- "SYSTEM I2C not configured" -- call SYSTEM I2C first
- "Classic not connected" -- I2C initialisation failed
- "Device connected is a Nunchuck" -- wrong device type

Reading Classic Controller Data: DEVICE()

```
value = DEVICE(WII parameter)
value = DEVICE(CLASSIC parameter)
```

Both WII and CLASSIC keywords are accepted interchangeably.

Parameter	Returns	Range	Description
LX	Integer	0-252	Left stick X
LY	Integer	0-252	Left stick Y
RX	Integer	0-248	Right stick X
RY	Integer	0-248	Right stick Y
L	Integer	0-248	Left trigger (analog)
R	Integer	0-248	Right trigger (analog)
B	Integer	0-32767	Button bitmask (15-bit)
T	Integer	Device type ID	

Classic Controller Button Bitmask

The button value from DEVICE(WII B) uses these bit positions (1 = pressed):

Bit	Value	Button
0	1	D-pad Right
1	2	D-pad Down
2	4	L Trigger (digital)
3	8	Select (-)
4	16	Home
5	32	Start (+)
6	64	R Trigger (digital)
7	128	(unused)
8	256	D-pad Left
9	512	D-pad Up
10	1024	ZR
11	2048	X
12	4096	A
13	8192	Y
14	16384	B

Example -- Classic Controller Game Input:

```
SYSTEM I2C GP4, GP5, 100000
```


Game Input Devices -- User Manual

```
WII CLASSIC OPEN ButtonPress

DO
  lx = DEVICE(WII LX)
  ly = DEVICE(WII LY)
  buttons = DEVICE(WII B)

  ' D-pad
  IF buttons AND 512 THEN dy = -1  ' Up
  IF buttons AND 2 THEN dy = 1    ' Down
  IF buttons AND 256 THEN dx = -1  ' Left
  IF buttons AND 1 THEN dx = 1     ' Right

  ' Face buttons
  IF buttons AND 4096 THEN fire = 1 ' A
  IF buttons AND 16384 THEN jump = 1 ' B
  ' ... render frame ...
LOOP

SUB ButtonPress
  LOCAL b = DEVICE(WII B)
  IF b AND 8 THEN ' Select
    paused = NOT paused
  END IF
END SUB
```

GPIO Button Input

For custom hardware -- arcade cabinets, handheld builds, or simple button panels -- you can wire physical buttons directly to GPIO pins. This works on all PicoMite builds with no external hardware beyond the buttons and optional pull-up resistors.

Setting Up Pins for Button Input

```
SETPIN pin, DIN [, PULLUP | PULLDOWN]
```

- pin -- physical pin number (1-44 on RP2040, 1-62 on RP2350) or GPIO notation (GP0, GP1, etc.)
- DIN -- digital input mode
- PULLUP -- enables internal ~50kOhm pull-up resistor (recommended for buttons wired to GND)
- PULLDOWN -- enables internal ~50kOhm pull-down resistor (for buttons wired to VCC)

Typical wiring: Connect one side of the button to a GPIO pin and the other to GND. Use PULLUP so the pin reads HIGH (1) when the button is released and LOW (0) when pressed.

Reading a Single Pin

```
value = PIN(pin)
```

Returns 0 or 1 for digital input pins. Reading is instantaneous -- a direct GPIO register read.

Example:

Game Input Devices -- User Manual

```
SETPIN GP2, DIN, PULLUP    ' button wired to GP2 and GND

DO
  IF PIN(GP2) = 0 THEN      ' 0 = pressed (active low with pull-up)
    PRINT "Button pressed!"
  END IF
LOOP
```

Reading Multiple Pins Atomically: PORT()

PORT() reads multiple consecutive pins in a single atomic snapshot -- all pin states are captured simultaneously. This is ideal for reading a bank of game buttons.

```
value = PORT(start_pin, nbits [, start_pin2, nbits2, ...])
```

- start_pin -- first pin in the range
- nbits -- number of consecutive pins to read
- Returns an integer with each bit representing a pin state (bit 0 = first pin)
- Multiple ranges can be concatenated

All pins in the range must be configured as DIN, DOUT, INTH, INTL, or INTB.

Example -- 4-Direction D-pad + Fire:

```
SETPIN GP2, DIN, PULLUP    ' Up
SETPIN GP3, DIN, PULLUP    ' Down
SETPIN GP4, DIN, PULLUP    ' Left
SETPIN GP5, DIN, PULLUP    ' Right
SETPIN GP6, DIN, PULLUP    ' Fire

DO
  buttons = PORT(GP2, 5)    ' read GP2-GP6 as 5-bit value
  ' Invert because pull-up = active low
  buttons = buttons XOR &b11111

  up    = buttons AND 1
  down  = (buttons >> 1) AND 1
  left  = (buttons >> 2) AND 1
  right = (buttons >> 3) AND 1
  fire  = (buttons >> 4) AND 1

  x = x + right - left
  y = y + down - up
  ' ... render frame ...
LOOP
```

Pin Interrupts

You can configure pins to trigger an interrupt on state changes, instead of polling every frame.

```
SETPIN pin, INTH, handler [, PULLUP | PULLDOWN]    ' low-to-high (rising edge)
SETPIN pin, INTL, handler [, PULLUP | PULLDOWN]    ' high-to-low (falling edge)
SETPIN pin, INTB, handler [, PULLUP | PULLDOWN]    ' both edges (any change)
```

- Up to 10 simultaneous pin interrupts

Game Input Devices -- User Manual

- Pin interrupts are software-pollled (checked between BASIC statements), not hardware edge-triggered. Latency depends on how fast the game loop executes.
- The interrupt fires once per detected transition -- the system saves the last pin state and compares it each poll cycle

For game buttons wired active-low with pull-up:

- Use INTL to trigger on button press (high-to-low)
- Use INTH to trigger on button release (low-to-high)
- Use INTB to trigger on both press and release

Example -- Interrupt-Driven Fire Button:

```
SETPIN GP6, INTL, FirePressed, PULLUP

' ... game loop runs without checking GP6 ...

SUB FirePressed
    fire_flag = 1
END SUB
```

SETTICK -- Timer-Based Polling

SETTICK creates a periodic timer interrupt for polling buttons at a fixed rate, providing natural debounce through the polling interval.

```
SETTICK period, handler [, tick_number]
SETTICK 0, 0 [, tick_number]           ' disable
SETTICK PAUSE [, tick_number]          ' pause
SETTICK RESUME [, tick_number]         ' resume
```

- period -- milliseconds between calls
- tick_number -- 1 to 4 (up to 4 independent timers)

Example -- 60Hz Button Polling:

```
DIM button_state
SETPIN GP2, DIN, PULLUP
SETTICK 16, PollButtons ' ~60Hz

' Main game loop uses button_state variable
DO
    ' ... use button_state for game logic ...
LOOP

SUB PollButtons
    button_state = NOT PIN(GP2) ' invert: 1 = pressed
END SUB
```

PIN(BOOTSEL) -- Built-in Button

Every Pico board has a BOOTSEL button. You can read it without any SETPIN configuration:

```
IF PIN(BOOTSEL) = 1 THEN PRINT "BOOTSEL pressed"
```

This reads the QSPI CS pin directly. Useful for a simple one-button input during development or testing.

Game Input Devices -- User Manual

Debounce Considerations

There is no built-in hardware debounce in the firmware. Mechanical buttons typically bounce for 5-20ms. Strategies:

1. Polling rate debounce -- Use SETTICK at 16-20ms intervals. The polling period naturally filters bounce since the button settles between samples.
2. Software debounce -- Read the pin, wait 10-20ms, read again. Only act if both reads agree.
3. Game loop debounce -- In a typical game loop running at 30-60 FPS, each frame is 16-33ms apart. Reading buttons once per frame provides natural debounce.
4. Edge detection -- Track the previous state and only act on transitions:

```
last_state = 1  ' pull-up = released = 1
DO
  current = PIN(GP2)
  IF current = 0 AND last_state = 1 THEN
    ' Button just pressed -- act on this frame only
    fire = 1
  END IF
  last_state = current
  ' ... render frame ...
LOOP
```

Touch Screen Input

PicoMite supports both resistive (SPI) and capacitive (I2C FT6336) touch screens, commonly used with SPI LCD displays. Touch input is ideal for menu selection, virtual buttons, drawing games, and drag-based controls.

Setup

Touch requires prior configuration of the system bus and a one-time calibration.

Resistive touch (SPI):

```
OPTION SYSTEM SPI GP18, GP19, GP16      ' CLK, MOSI, MISO
OPTION TOUCH cs_pin, irq_pin [, click_pin]
```

Capacitive touch (I2C FT6336):

```
OPTION SYSTEM I2C GP14, GP15            ' SDA, SCL
OPTION TOUCH FT6336 irq_pin, addr_pin [, click_pin] [, threshold]
```

Disable:

```
OPTION TOUCH DISABLE
```

After configuring, calibrate the touch screen:

```
GUI CALIBRATE
```

This runs an interactive 4-point calibration that maps raw touch coordinates to screen pixels. Calibration is saved to flash.

You can also set calibration values manually:

```
GUI CALIBRATE swapxy, xzero, yzero, xscale, yscale
```

Game Input Devices -- User Manual

TOUCH() Function -- Reading Touch Coordinates

Syntax:

```
x = TOUCH(X)           ' current X coordinate (-1 if no touch)
y = TOUCH(Y)           ' current Y coordinate (-1 if no touch)
d = TOUCH(DOWN)        ' 1 if pen/finger is touching, 0 if not
u = TOUCH(UP)          ' 1 if pen/finger is lifted, 0 if touching
x = TOUCH(LASTX)       ' X when pen was last lifted
y = TOUCH(LASTY)       ' Y when pen was last lifted
r = TOUCH(REF)         ' GUI control number being touched
r = TOUCH(LASTREF)     ' last GUI control that was touched
```

Capacitive-only (multi-touch):

```
x2 = TOUCH(X2)         ' second touch point X
y2 = TOUCH(Y2)         ' second touch point Y
```

TOUCH(X) and TOUCH(Y) return -1 when no touch is detected.

GUI INTERRUPT -- Touch-Driven Interrupts

GUI INTERRUPT fires when a touch-down or touch-up event occurs, enabling interrupt-driven touch handling without polling.

Syntax:

```
GUI INTERRUPT down_handler [, up_handler]
GUI INTERRUPT 0           ' disable
```

- down_handler -- called when the screen is touched
- up_handler -- called when the touch is released

Inside the handler, use TOUCH(X), TOUCH(Y) etc. to read the coordinates.

GUI TEST TOUCH

Interactive diagnostic -- draws dots wherever you touch the screen. Press Enter to exit.

```
GUI TEST TOUCH
```

Example -- Virtual Buttons for a Game

```
' Draw touch zones
BOX 0, 180, 80, 60, 2, RGB(WHITE), RGB(64,64,64)
TEXT 40, 210, "LEFT", "CM"
BOX 80, 180, 80, 60, 2, RGB(WHITE), RGB(64,64,64)
TEXT 120, 210, "RIGHT", "CM"
BOX 240, 180, 80, 60, 2, RGB(WHITE), RGB(RED,0,0)
TEXT 280, 210, "FIRE", "CM"

' Game loop
DO
  dx = 0 : fire = 0
  IF TOUCH(DOWN) THEN
    tx = TOUCH(X) : ty = TOUCH(Y)
    IF ty >= 180 THEN
```

Game Input Devices -- User Manual

```
IF tx < 80 THEN dx = -1          ' Left zone
IF tx >= 80 AND tx < 160 THEN dx = 1 ' Right zone
IF tx >= 240 THEN fire = 1       ' Fire zone
END IF
END IF
x = x + dx * speed
' ... render frame ...
LOOP
```

Example -- Drag Control

```
DIM prev_tx = -1, prev_ty = -1

DO
  IF TOUCH(DOWN) THEN
    tx = TOUCH(X) : ty = TOUCH(Y)
    IF prev_tx <> -1 THEN
      dx = tx - prev_tx
      dy = ty - prev_ty
      ' Move game object by drag delta
      obj_x = obj_x + dx
      obj_y = obj_y + dy
    END IF
    prev_tx = tx : prev_ty = ty
  ELSE
    prev_tx = -1 : prev_ty = -1 ' Reset on lift
  END IF
  ' ... render frame ...
LOOP
```

Tips for Game Development with Touch

- Make touch zones large -- at least 40x40 pixels for comfortable finger targets on small LCDs.
- Use TOUCH(DOWN) for continuous actions (movement), TOUCH(LASTX)/TOUCH(LASTY) for tap-and-release actions (menu selection).
- Capacitive screens (FT6336) support two-finger multi-touch via TOUCH(X2), TOUCH(Y2) -- useful for pinch/rotate gestures.
- Debounce taps -- a quick PAUSE 50 after detecting a tap prevents double-triggering.
- GUI controls (BUTTON, SWITCH, AREA) can be used alongside direct TOUCH() calls for hybrid interfaces -- the GUI system handles hit detection and visual feedback automatically.

Interrupt Priority

When multiple input interrupts are active, MMBasic processes them in this priority order (highest first):

1. Touch screen
2. CFunction interrupts
3. ON KEY (keyboard)
4. I2C Slave

Game Input Devices -- User Manual

5. COM (serial) interrupts
6. GUI interrupts
7. WAV (audio playback)
8. IR (infrared)
9. I/O Pin interrupts (INTH/INTL/INTB, in definition order)
10. Tick interrupts (SETTICK 1-4)

Wii controller and USB Gamepad interrupts are processed via the nunstruct/nunfoundc mechanism at a priority comparable to pin interrupts.

Only one interrupt can be active at a time. A pending interrupt will wait until the current handler returns via END SUB or IRETURN.

Choosing the Right Input Method

For Simple Games (Any Build)

Use INKEY\$ in the game loop for basic keyboard controls, or GPIO buttons with PIN()/PORT() for custom hardware. These work everywhere with no special build requirements.

For Action Games (USB Builds)

Use KEYDOWN for keyboard-based action games -- it detects up to 6 simultaneous keys and modifier states. Or use a USB Gamepad for the best control experience with analog sticks and many buttons.

For Action Games (Non-USB Builds)

Use ON PS2 with MM.PS2 to track key-down and key-up events from a PS/2 keyboard. This enables simultaneous key detection without requiring a USB build.

For Motion/Tilt Games (Any Build)

Use the Wii Nunchuck -- its 3-axis accelerometer enables tilt controls, and the joystick provides analog movement. Requires only I2C wiring.

For Console-Style Games (Any Build)

The Wii Classic Controller or a USB Gamepad provides the full console experience with dual sticks, triggers, and a d-pad. The Classic works on any build via I2C; USB gamepads require a USB build.

For Touch-Screen Games (LCD Builds)

Use TOUCH() to read tap and drag coordinates directly. Define virtual button zones on screen, or use GUI INTERRUPT for event-driven touch handling. Works with both resistive (SPI) and capacitive (I2C) touch panels.

For Arcade Cabinets

Use GPIO buttons with PORT() for atomic multi-button reads. Wire buttons between GPIO pins and GND with internal

Game Input Devices -- User Manual

PULLUP. Use SETTICK for fixed-rate polling with natural debounce.

Complete Example: Multi-Input Game Template

This example shows a game loop that supports both keyboard (KEYDOWN) and USB Gamepad input simultaneously:

```
' Initialise display
MODE 2
FRAMEBUFFER CREATE
FRAMEBUFFER WRITE F

' Input state variables
DIM dx, dy, fire, jump

' Detect gamepad
DIM gp_channel = 0
FOR ch = 1 TO 4
  IF DEVICE(GAMEPAD ch, "T") > 0 THEN
    gp_channel = ch
    EXIT FOR
  END IF
NEXT ch

' Game loop
DO
  dx = 0 : dy = 0 : fire = 0 : jump = 0

  ' Read keyboard (KEYDOWN - USB builds)
  n = KEYDOWN(0)
  FOR i = 1 TO n
    k = KEYDOWN(i)
    IF k = 128 THEN dy = dy - 1    ' Up
    IF k = 129 THEN dy = dy + 1    ' Down
    IF k = 130 THEN dx = dx - 1    ' Left
    IF k = 131 THEN dx = dx + 1    ' Right
    IF k = 32 THEN fire = 1        ' Space
    IF k = ASC("z") OR k = ASC("Z") THEN jump = 1
  NEXT i

  ' Read gamepad (if connected)
  IF gp_channel > 0 THEN
    LOCAL b = DEVICE(GAMEPAD gp_channel, "B")
    IF b AND 128 THEN dy = dy - 1    ' Up
    IF b AND 32 THEN dy = dy + 1    ' Down
    IF b AND 256 THEN dx = dx - 1    ' Left
    IF b AND 64 THEN dx = dx + 1    ' Right
    IF b AND 8192 THEN fire = 1      ' Cross/A
    IF b AND 4096 THEN jump = 1      ' Square/X

    ' Add analog stick input with deadzone
    LOCAL lx = DEVICE(GAMEPAD gp_channel, "LX")
    LOCAL ly = DEVICE(GAMEPAD gp_channel, "LY")
    IF ABS(lx - 128) > 20 THEN dx = dx + SGN(lx - 128)
```


Game Input Devices -- User Manual

```
IF ABS(ly - 128) > 20 THEN dy = dy + SGN(ly - 128)
END IF

' Clamp
IF dx > 1 THEN dx = 1
IF dx < -1 THEN dx = -1
IF dy > 1 THEN dy = 1
IF dy < -1 THEN dy = -1

' Update game state
player_x = player_x + dx * speed
player_y = player_y + dy * speed

' Render
CLS
' ... draw game world ...
FRAMEBUFFER COPY F, N

' Frame timing
PAUSE 16
LOOP
```

Further Reading

Document	Description
Game_Development_Guide.pdf	Comprehensive game development overview
SPRITE_User_Manual.pdf	Sprite engine reference
TILEMAP_User_Manual.pdf	Tile map engine reference
BLIT_User_Manual.pdf	Block image transfer reference
PLAY_SAMPLE_User_Manual.pdf	Wavetable audio synthesis