Command:EXTRA (EX)

MS2000 and RM2000 Syntax

Shortcut	EX *version 9.53*
Format	EXTRA [X?] [Y?] [Z=lock_ki] [M=button_code] [R=small_enc] [T?]
Remembered	Using SS Z

Tiger Syntax

Shortcut	EX *version 3.51*
Format	[Addr#]EXTRA [X?] [Y?] [Z=lock_ki] [M=button_code] [R=small_enc] [T?]
Type	Card-Addressed
Remembered	Using [Addr#]SS Z

X? Provides the `CRISP` bottom line string as is shown on the LCD display.

Y? Returns the SNR value shown on the LCD after log amp calibration.

The **Z** argument sets the integral error servo parameter. The default is `1`. Higher values may improve speed settling but can also generate instability. Use sparingly.

This is also the `lock_ki` value for `CRISP`. When `CRISP` enters the lock state (`LK F=83`) it changes the KI Z value for the Z axis. `KI Z` is restored to the initial value when `CRISP` enters the stop state (`LK F=79`).

When CRISP restores KI Z after using lock_ki, it uses a saved value that is set only once when the controller powers on. If you want to change KI Z, use SS Z and power cycle the controller so that it can restore KI Z to the correct value. It is not recommended to set KI Z unless you are an advanced user.

MS-2000 9.2p or Tiger v3.42 required

T?: The controller detects the resolution of the ADC during initialization.

Code	DAC	CPU
0	10-bit ADC	`C8051F122`
1	12-bit ADC	`C8051F120`

MS-2000 9.2n or Tiger v3.36 required

M? Returns the `button_flag_byte` and resets the value to `0`.

M=# Modify the `button_flag_byte` with the button code and call the button functions associated with that code.

This command differs from BE F which does not modify the `button_flag_byte` and only calls a single button function.

Additional Details About M?

The `button_flag_byte` stores the state of the last detected button press for each button. When the controller is powered on, the value is initialized to `0`. As the user presses buttons the value of the `button_flag_byte` changes, it is important to note that this value only changes when you **release** the button.

After receiving the `EXTRA M?` command, the internal value on the controller is reset to `0`, enabling you to detect new button presses.

If a button has already been pressed, and then is pressed again, the new state overwrites the old state for that button. Example: if you do a `Normal Press` and then a `Long Press` on the `Joystick Button`, the next time you send the "`EXTRA M?`" command the state of the `Joystick Button` will be `Long Press`.

Zero/Halt button presses only have the states 0 and 1. (Not Pressed and Normal Press)

The `button_flag_byte` is divided into four `2-bit` sections that each contain the state of a button:

Bits	Button
1-2	@ Button
3-4	Home Button
5-6	Joystick Button
7-8	Zero/Halt Button

Each `2-bit` section can take on the values `0-3`, these codes represent the state of the button.

Decimal	Binary	State
0	`00`	Not Pressed
1	`01`	Normal Press
2	`10`	Long Press
3	`11`	Extra Long Press

Example:

1. @ Button Normal Press
2. Home Button Long Press
3. Joystick Extra Long Press
4. Zero/Halt Normal Press
5. Send serial command `EXTRA M?`

Results of steps `1-5` in binary:

1. `button_flag_byte = 00 00 00 01`
2. `button_flag_byte = 00 00 10 01`
3. `button_flag_byte = 00 11 10 01`
4. `button_flag_byte = 01 11 10 01`
5. `button_flag_byte = 00 00 00 00` `(serial command reset)`

Example Python code for extracting button states from the button_flag_byte:

Python Parse Button Flag

asi_parse_button_flag.py

```python
# the value returned from EXTRA M?
button_flag_byte = 127

# bit masks
mask_at   = 0x03 # 00000011
mask_home = 0x0C # 00001100
mask_js   = 0x30 # 00110000
mask_zero = 0xC0 # 11000000

# get the button states from button_flag_byte
btn_at   = button_flag_byte & mask_at
btn_home = (button_flag_byte & mask_home) >> 2
btn_js   = (button_flag_byte & mask_js) >> 4
btn_zero = (button_flag_byte & mask_zero) >> 6

# show the results in decimal and binary
print(f"{button_flag_byte = } (binary {button_flag_byte :08b})")
print(f"{btn_at = } (binary {btn_at:02b})")
print(f"{btn_home = } (binary {btn_home:02b})")
print(f"{btn_js = } (binary {btn_js:02b})")
print(f"{btn_zero = } (binary {btn_zero:02b})")

# console output:
# button_flag_byte = 127 (binary 01111111)
# btn_at = 3 (binary 11)
# btn_home = 3 (binary 11)
# btn_js = 3 (binary 11)
# btn_zero = 1 (binary 01)
```

Additional Details About M=button_code

This function allows you to simulate button presses programmatically through a serial command.

This command modifies the `button_flag_byte` and calls the button functions associated with that button code.

The button codes are the same values that are returned by `EXTRA M?`. The input value is clamped to the range: 0-127.

If a button code represents multiple button presses then the button functions will be called in

the order ⇒
@, Home, Joystick, Zero/Halt (LSB ⇒ MSB)

You can expect the same behavior as if you were pressing physical buttons ⇒

1. Send the command `EXTRA M=3`: `button_flag_byte` = 3, @ Extra Long Press button function called.
2. Send the command `EXTRA M=1`: `button_flag_byte` = 1, @ Normal Press button function called.
3. Send the command `EXTRA M=5`: `button_flag_byte` = 5, @ Normal Press and Home Normal Press button functions called.

This demonstrates that button presses are overwritten as if you were interacting with the physical controller pressing buttons.

Example Python code for creating a button_flag_byte:

Python Create Button Flag

asi_create_button_flag.py

```
def create_button_code(at: int = 0, home: int = 0, joystick: int =
0, zero_halt: int = 0) -> int:
    assert at in range(4), "Must be in the range 0-3."
    assert home in range(4), "Must be in the range 0-3."
    assert joystick in range(4), "Must be in the range 0-3."
    assert zero_halt == 0 or zero_halt == 1, "Must be 0 or 1."

    button_code = 0

    # bit masks
    BITMASK_AT   = 0x03 # 00000011
    BITMASK_HOME = 0x0C # 00001100
    BITMASK_JS   = 0x30 # 00110000
    BITMASK_ZERO = 0xC0 # 11000000

    # set bits
    button_code &= ~BITMASK_AT
    button_code |= at & BITMASK_AT

    button_code &= ~BITMASK_HOME
    button_code |= (home << 2) & BITMASK_HOME

    button_code &= ~BITMASK_JS
    button_code |= (joystick << 4) & BITMASK_JS

    button_code &= ~BITMASK_ZERO
    button_code |= (zero_halt << 6) & BITMASK_ZERO
    return button_code

def main():
    button_code = create_button_code(at=1, home=1)
    print(button_code)
    # prints 5

if __name__ == "__main__":
    main()
```

commands, tiger, ms2000, crisp