

## Product Datasheet: MCI-100P

### *Microcontroller/CAN Interface for use with the Parallax BASIC Stamp®*

Revised: March 8, 2005

---

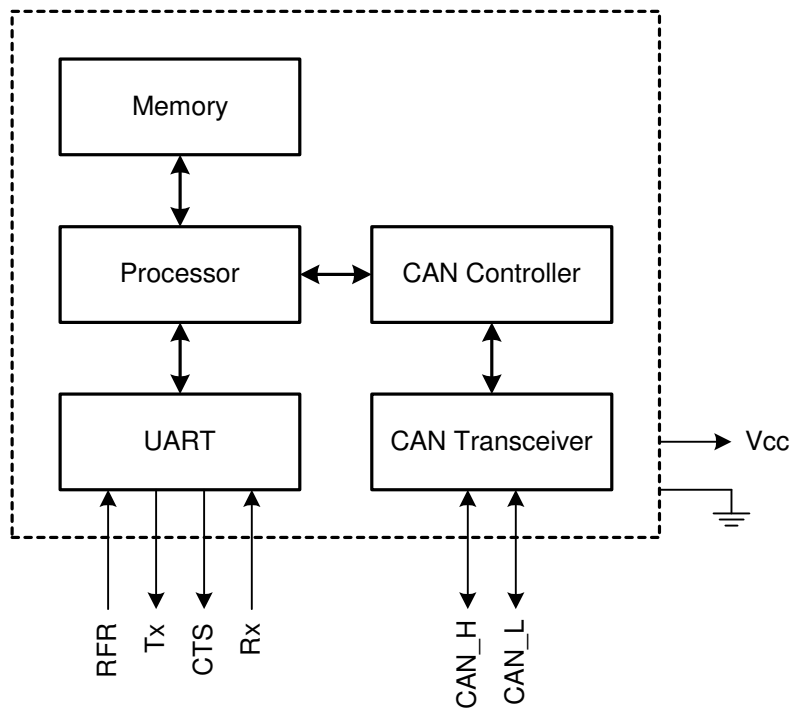
## Features

- Designed specifically for the Parallax BASIC Stamp®
  - Minimizes the memory requirements and processing burden on the Stamp
  - Maximizes the data transfer rate
  - Can be powered off of the Stamp's on-board power regulator
  - Field programmable to support future changes and enhancements
- Controller Area Network (CAN) port
  - CAN 2.0A & 2.0B compliant
  - Compatible with the ISO 11898-2 standard
  - High speed (up to 1 Mbit/sec)
  - Supports both 11-bit and 29-bit identifiers
- UART port
  - Communication speeds up to 115,200 bits/sec
  - Direct UART to UART connection. No transceiver necessary
- 240 byte internal receive buffer

## Description

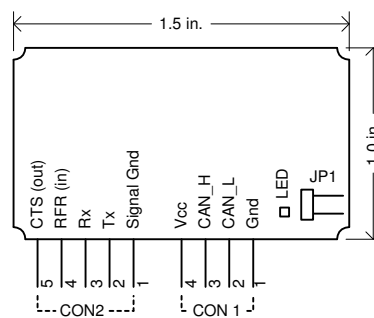
The MCI-100P Microcontroller/CAN Interface is a low-power, high-performance CAN co-processor designed for use with the Parallax BASIC Stamp®. The MCI-100P minimizes the memory requirements and processing burden on the Stamp while maximizing the data transfer rate. The low-level CAN hardware details are encapsulated in an easy-to-use, PBasic software interface. The MCI-100P has the capability of being reprogrammed in the field, making it possible to support custom applications and higher layer CAN protocols such as CANopen and DeviceNet.

## Block Diagram



**Figure 1.** Block Diagram

## Pin Configuration



**Figure 2.** Pinout MCI-100P

## Pin Descriptions

Pin	Name	Description
1	Gnd	External power ground
2	CAN_L	CAN_L bus signal (dominant low)
3	CAN_H	CAN_H bus signal (dominant high)
4	Vcc	External power Vcc (+5V)

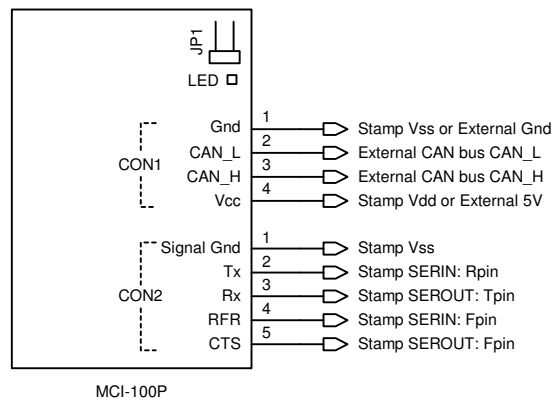
**Table 1. Power/CAN Connector (CON 1)**

Pin	Name	Description
1	Signal Gnd	Signal ground
2	Tx	Transmitted data
3	Rx	Received data
4	RFR (in)	Ready-For-Receive (input from Stamp)
5	CTS (out)	Clear-To-Send (output to Stamp)

**Table 2. UART Connector (CON 2)**

## Hardware Operation

The MCI-100P has a simple hardware interface. It can be connected directly to the BASIC Stamp, requiring no additional electronic components. It is possible to power the device directly off of the BASIC stamp or from any +5 VDC power source.



**Figure 3. Interfacing the MCI-100P with the BASIC Stamp and a CAN bus**

## Communication between the MCI-100P and the BASIC Stamp

The MCI-100P and the BASIC Stamp communicate via a serial interface. The PBasic commands SERIN and SEROUT are used for this purpose. The MCI-100P uses hardware flow control to prevent overrunning the BASIC Stamp. The flexibility of the SERIN and SEROUT commands give you the choice of which pins you would like to use for receive, transmit and flow control (see the PBasic manual for more information). The following table summarizes how the connections should be made.

MCI-100P Pin	Basic Stamp Pin
CON2: Pin 1 (Signal Ground)	Vss
CON2: Pin 2 (Tx)	SERIN: Rpin
CON2: Pin 3 (Rx)	SEROUT: Tpin
CON2: Pin 4 (RFR)	SERIN: Fpin
CON2: Pin 5 (CTS)	SEROUT: Fpin

**Table 3.** MCI-100P to BASIC Stamp UART connections

The MCI-100P supports the following PBasic baud rates: 600, 1200, 2400, 4800, 9600, 19200 and 38400.

## Powering the MCI-100P

The MCI-100P requires a regulated +5 VDC power source. It is possible to power the device from the BASIC Stamp or from an external supply.

When powering from the BASIC Stamp, the Gnd and Vcc pins of the MCI-100P should be connected to the Stamp's Vss and Vdd pins, respectively. It is recommended that the status LEDs of the MCI-100P be disabled when powering off of the Stamp's power regulator (see the "Disable LEDs" function below).

When powering from an external power source, make certain that there is only one supply.

MCI-100P Pin	Basic Stamp Pin or External Supply
CON1: Pin 1 (Gnd)	Vss or ground of the external supply
CON1: Pin 4 (Vcc)	Vdd or +5VDC of the external supply

**Table 4.** MCI-100P to BASIC Stamp power connections

## Interfacing with a CAN bus

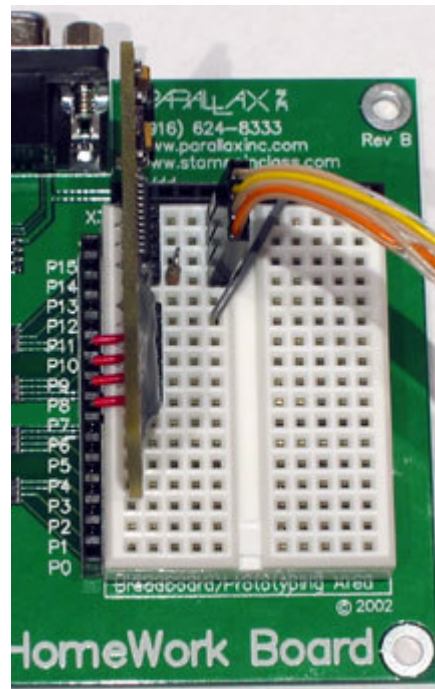
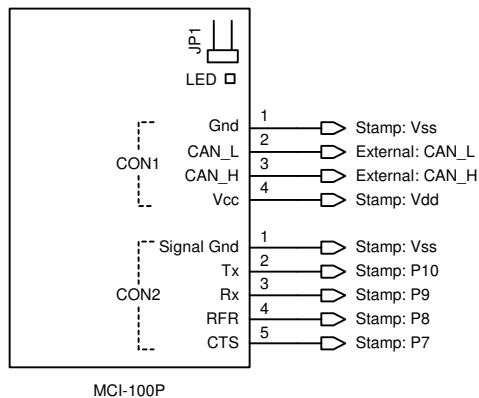
Connecting the MCI-100P to a CAN bus is straightforward. The MCI-100P's CAN\_L and CAN\_H lines should be connected to the CAN\_L and CAN\_H line of the CAN bus, respectively. Please note, if the MCI-100P is the last device on either end of the bus, a 120 ohm resistor should be used to terminate the bus. The resistor should connect CAN\_H and CAN\_L together. It is always a good idea to properly terminate the bus, even though it may not seem necessary with short bus lengths.

MCI-100P Pin	CAN bus
CON1: Pin 2 (CAN_L)	CAN_L
CON1: Pin 3 (CAN_H)	CAN_H

**Table 5.** MCI-100P to CAN bus connections

### **Example: The BASIC Stamp HomeWork Board**

The diagram and picture in Figure 4 demonstrate one method of connecting the MCI-100P to the BASIC Stamp HomeWork Board. In this example the MCI-100P is powered off of the BASIC Stamp's power regulator. Also, note the terminating resistor located between the CAN bus cable and the MCI-100.



**Figure 4:** The MCI-100P connected to the BASIC Stamp HomeWork Board

## Software Operation

If you are not familiar with Controller Area Network, we recommend that you read the document “An Overview of Controller Area Network (CAN) Technology”. The document can be found on the Machine Bus web site at <http://www.machinebus.com/documents/canTechnologyOverview.pdf>.

The MCI-100P and the firmware loaded on it have been designed with the following goals in mind:

- **Minimize the memory requirements and processing burden on the Stamp.** CAN has the potential of placing huge demands on a processor. The MCI-100P is a microcontroller-based solution. The advantage to this design is that the MCI-100P becomes a CAN co-processor for the Stamp. This solution relieves the Stamp of much of the memory and processing requirements needed to manage CAN communications.
- **Maximize the data transfer rate between the MCI-100P and the Stamp.** The Stamp has the highest data transfer rate using asynchronous serial communication<sup>1</sup>. The MCI-100P is capable of communicating at speeds up to 115,200 baud.
- **Make the MCI-100P re-programmable to support future enhancements.** The MCI-100P uses a field-programmable flash and eeprom memory. The ability to reprogram the MCI-100P in the field opens up a number of possibilities, including the ability to tailor the interface to a particular application or to support common higher layer protocols such as CANopen and DeviceNet (contact Machine Bus for more information).

## The Basic CAN Controller API

All aspects of the CAN protocol are typically implemented by hardware based CAN controllers. This reduces the load on the host system that requires access to the CAN bus. These controllers are typically implemented as stand-alone integrated circuits or as part of a microcontroller package. There are two primary classes of CAN controller, Basic CAN and Full CAN. The acceptance filtering and buffering capabilities are what differentiate these two controller classes.

A CAN system is based on the broadcast principle. This means that all nodes receive every message that is transmitted on the bus. In most cases a receiving node will only be interested in a subset of the transmitted messages. Acceptance filtering is used to limit the number of messages that the host system needs to deal with. Typically the acceptance filter is only concerned with the message identifier but other methods of filtering maybe possible depending upon the particular controller's capabilities.

Basic CAN controllers are characterized as having one transmit buffer and one receive buffer with a single acceptance filter. Some Basic CAN controllers are capable of storing multiple messages in its buffer, but the same acceptance filter is used on all incoming messages. The Basic CAN approach is easy to use and is available on most all controller implementations.

The MCI-100P presents an API that models the Basic CAN controller. The software interface is relatively simple and designed to reduce the memory and processing burden on the stamp. Because all interaction with the MCI-100P must take place via serial communication, the Basic CAN API consists entirely of data structures. These data structures represent functions, their return values and information used by your BASIC Stamp application.

---

<sup>1</sup> According to the Stamp PBasic manual

## Calling MCI-100P Functions

The MCI-100P communicates with the BASIC Stamp via a serial interface. Therefore, all commands sent to and all results received from the MCI-100P are in the form of a data stream. The structure of a command follows a format very similar to a function in the C language. The basic pattern is as follows:

1. **Transmit the command code.** A command begins by transmitting a command code to the MCI-100P. The command code is a single byte that instructs the MCI-100P which operation you are invoking.
2. **Transmit any parameters.** If a command accepts parameters, then these are transmitted next. The MCI-100P expects that all parameters will be sent and will not process the command until it receives them. Special care must be taken when sending data structures with variable lengths. The CAN Message is a good example. There are two variables encoded into the "fieldInfo" field of the CAN Message that determine its length. The MCI-100P will be expecting, and will only accept, the amount of data specified by this field.
3. **Receive any resulting data.** If a command returns any resulting data, then the MCI-100P will transmit it to the BASIC Stamp after the last parameter byte is received. The MCI-100P will not accept another command until the BASIC Stamp receives all the data. Again, special care must be taken when accepting data structures with a variable length.
4. **Receive the status code.** Every command ends with a status code. The status code is sent by the MCI-100P to indicate whether the command was successful or not.

## The MCI-100P Boot Process

There are some start up activities that need to take place before the MCI-100P is ready to receive and transmit CAN messages. The PBasic example in Appendix A demonstrates how the boot process is executed. The following describes the process.

1. **Establish the serial data rate.** The data rate between the BASIC Stamp and the MCI-100P needs to be established. The MCI-100P is able to automatically determine the speed the Stamp is transmitting at. Sending a single ASCII carriage return character to the MCI-100P at the intended data rate will accomplish this. The MCI-100P supports the following Stamp baud rates: 600, 1200, 2400, 4800, 9600, 19200 and 38400.
2. **Enter command state.** At this point, the communication between the BASIC Stamp and the MCI-100P should be established. To test this, and to confirm that the MCI-100P is ready to accept commands, a series of fourteen (14) null values should be sent to the MCI-100P.
3. **Initialize the MCI-100P.** An "Initialize" command should be sent to the MCI-100P.
4. **Set the CAN bit timings.** Bit timings are used by the CAN protocol to establish the parameters for communication on the CAN bus. They are relatively complicated to determine. There are two ways to set the CAN bit timings on the MCI-100P. Either the "Set Bit Rate" or the "Set Bit Timings" command can be used. The "Set Bit Timings" allows the caller to set the raw bit timing values. This command should only be used in extreme circumstances. The "Set Bit Rate" command will automatically set the proper CAN bit timings, based on the data rate requested (e.g. 125 kbits/sec).
5. **Set the CAN acceptance filter.** A CAN system is based on the broadcast principle. This means that all nodes receive every message that is transmitted on the bus. To keep from overwhelming the BASIC Stamp with unnecessary messages, an acceptance filter can be set. The acceptance filter is a feature of the Basic CAN controller and is implemented in hardware, so it is very efficient. There are two ways to set the acceptance filter in the MCI-100P. Either the "Set Filter" or the "Receive All" command can be used. The "Set Filter" command allows you to set a very specific CAN acceptance filter, but is rather complicated. As a shortcut, the "Receive All" command is provided to automatically configure the acceptance filter to receive all CAN messages.

6. **Enable the MCI-100P.** Once the MCI-100P has been properly configured, it needs to be enabled. An "Enable" command allows it to take place in CAN bus communications.
7. **Enter a message processing loop.** Once the MCI-100P has been enabled, its status must be polled continuously. The BASIC Stamp should enter an endless "DO" loop where it calls and processes the "Status" command. All other application tasks should also take place in with this loop.

## Data: CAN Message

---

**Description:** This is a true data structure and represents a CAN message. A CAN message is a packet of information that is carried on a CAN bus. CAN Messages consist of header information and a data “payload”. A CAN message is intentionally designed to carry a maximum payload of eight (8) data bytes.

Table 6 describes the structure of a CAN Message. Note that this is a variable length structure whose size can vary from 3 to 13 bytes in length.

Field Size (Bytes)	Field Name	Description
1	FrameInfo	Contains information on the frame format, the frame type, and the data length (see Figure 11)
2 or 4	CAN identifier	Either an 11-bit or 29-bit CAN identifier
0 – 8	CAN data	Data length can vary from 0 to 8 bytes

**Table 6:** *FrameInfo Field Format*

The size of the CAN identifier and the CAN data is contained within the FrameInfo byte. Table 7 describes the structure of the FrameInfo Byte.

Field Size (Bits MSB to LSB)	Field Name	Description
8	Frame Format	0 = Standard Frame (11-bit id) or, 1 = Extended Frame (29-bit id)
7	Remote Request	0 = Data Frame or, 1 = Remote Frame (Does not contain data)
5 – 6	reserved	Must be zero
1-4	Data length	The number of data bytes

**Table 7:** *FrameInfo Field Format*

### Example:

The following snippet of PBasic code demonstrates how to format a CAN Message for debug output.

```
'CAN message constants
FrameInfoExtended CON 128 'Frame format bit
FrameInfoRemote CON 64 'Remote transmission request bit
FrameInfoDataLength CON 15 'Data length code mask

'Variables
canMessage VAR Byte(13)
index VAR Byte
length VAR Nib

'Print the FrameInfo as a hexadecimal number
DEBUG "0x", HEX2 canMessage( 0 ), " " 'FrameInfo

'Determine whether this message contains a standard or an extended id
IF( canMessage( 0 ) & FrameInfoExtended ) THEN Print_Extended_Id

'Print the standard id as a hexadecimal number
Print_Standard_Id:
DEBUG "0x0000"
```

```

DEBUG HEX2 canMessage( 1 ) 'MSB of Standard Id
DEBUG HEX2 canMessage( 2 ) 'LSB of Standard Id
index = 3
GOTO Print_Message_Data

'Print the extended id as a hexadecimal number
Print_Extended_Id:
DEBUG "0x"
DEBUG HEX2 canMessage( 1 ) ' MSB of Extended Id
DEBUG HEX2 canMessage( 2 )
DEBUG HEX2 canMessage( 3 )
DEBUG HEX2 canMessage( 4 ) ' LSB of Extended Id
index = 5

'Print the data payload if this is not a remote frame
Print_Message_Data:
DEBUG " "
IF( canMessage( 0 ) & FrameInfoRemote ) THEN End_Of_Print
length = ( index - 1 + ( canMessage( 0 ) & FrameInfoDataLength ) )
FOR index = index TO length
    DEBUG HEX2 canMessage( index ), " "
NEXT

End_Of_Print:
DEBUG CR

```

## Function: Set Bit Timings

---

**Description:** Sets the CAN controller's bit timing parameters (see the 'Bit Rate' command as an alternative)

**Parameters:** 7 bytes in length

Field Size (Bytes)	Field Name	Field Description
1	Command Code	Always equal to '1' for this command
1	Baud Rate Prescaler	Divisor for the CAN controller's timing source
1	Propagation Delay Segment	Length of the Propagation Delay Segment in time quanta
1	Phase Buffer Segment 1	Length of Phase Buffer Segment 1 in time quanta
1	Phase Buffer Segment 2	Length of Phase Buffer Segment 2 in time quanta
1	Synchronization Jump Width	Length of the Synchronization Jump Width in time quanta
1	Sample Mode	0 for one sample; 1 for three samples

**Returns:** Zero if successful.

### Examples:

The following snippet of PBasic code demonstrates how to set the CAN bit timing to 125 kbits/sec.

```
'CAN controller commands
CommandBitTiming CON 1

'Serial communication parameters
SerialBaudmode CON 6
SerialCts      PIN 7
SerialRfr      PIN 8
SerialTx       PIN 9
SerialRx       PIN 10

'CAN bit timing settings
BaudRatePrescaler CON 4
PropagationDelay   CON 7
PhaseBuffer1       CON 7
PhaseBuffer2       CON 7
SynchronizationJumpWidth CON 1
SampleMode         CON 1

'Variables
status VAR Byte

'Execute the set bit timing function
SEROUT SerialTx\SerialCts, SerialBaudmode, [CommandBitTiming]
SEROUT SerialTx\SerialCts, SerialBaudmode, [BaudRatePrescaler]
SEROUT SerialTx\SerialCts, SerialBaudmode, [PropagationDelay]
SEROUT SerialTx\SerialCts, SerialBaudmode, [PhaseBuffer1]
SEROUT SerialTx\SerialCts, SerialBaudmode, [PhaseBuffer2]
SEROUT SerialTx\SerialCts, SerialBaudmode, [SynchronizationJumpWidth]
SEROUT SerialTx\SerialCts, SerialBaudmode, [SampleMode]

'Check the return value
SERIN SerialRx\SerialRfr, SerialBaudmode, [status]
```

## Function: Disable

---

**Description:** Disables the CAN controller. The CAN controller does not participate in bus activities when it is disabled.

**Parameters:** 1 byte in length

Field Size (Bytes)	Field Name	Field Description
1	Command Code	Always equal to '2' for this command

**Returns:** Zero if successful.

### Examples:

The following snippet of PBasic code demonstrates how to disable the CAN controller.

```
'CAN controller commands
CommandDisable CON 2

'Serial communication parameters
SerialBaudmode CON 6
SerialCts      PIN 7
SerialRfr      PIN 8
SerialTx       PIN 9
SerialRx       PIN 10

'Variables
status VAR Byte

SEROUT SerialTx\SerialCts, SerialBaudmode, [CommandDisable]

'Check the return value
SERIN SerialRx\SerialRfr, SerialBaudmode, [status]
```

## Function: Enable

---

**Description:** Enables the CAN controller. The CAN controller participates in bus activities when it is enabled.

**Parameters:** 1 byte in length

Field Size (Bytes)	Field Name	Field Description
1	Command Code	Always equal to '3' for this command

**Returns:** Zero if successful.

### Examples:

The following snippet of PBasic code demonstrates how to enable the CAN controller.

```
'CAN controller commands
CommandEnable CON 3

'Serial communication parameters
SerialBaudmode CON 6
SerialCts      PIN 7
SerialRfr      PIN 8
SerialTx       PIN 9
SerialRx       PIN 10

'Variables
status VAR Byte

SEROUT SerialTx\SerialCts, SerialBaudmode, [CommandEnable]

'Check the return value
SERIN SerialRx\SerialRfr, SerialBaudmode, [status]
```

## Function: Get Error Count

---

**Description:** Returns the status of the CAN controller's error registers

**Parameters:** 1 byte in length

Field Size (Bytes)	Field Name	Field Description
1	Command Code	Always equal to '4' for this command

**Returns:** 3 bytes

Field Size (Bytes)	Field Name	Field Description
1	Receive Error Count	CAN controller's receive error count
1	Transmit Error Count	CAN controller's transmit error count
1	Call Status Code	Zero if successful

## Examples:

The following snippet of PBasic code demonstrates how to retrieve the state of the error count registers.

```
'CAN controller commands
CommandErrorCount CON 4

'Serial communication parameters
SerialBaudmode CON 6
SerialCts      PIN 7
SerialRfr      PIN 8
SerialTx       PIN 9
SerialRx       PIN 10

'Variables
rxErrorCount VAR Byte
status       VAR Byte
txErrorCount VAR Byte

SEROUT SerialTx\SerialCts, SerialBaudmode, [CommandErrorCount]

SERIN SerialRx\SerialRfr, SerialBaudmode, [rxErrorCount]
SERIN SerialRx\SerialRfr, SerialBaudmode, [txErrorCount]

DEBUG "Receive error count = ", rxErrorCount
DEBUG " , Transmit error count", txErrorCount, CR

'Check the return value
SERIN SerialRx\SerialRfr, SerialBaudmode, [status]
```

## Function: Set Filter

---

**Description:** Sets the CAN controller's acceptance filter (see the 'Receive All' command as an alternative).

**Parameters:** 13 bytes in length

Field Size (Bytes)	Field Name	Field Description
1	Command Code	Always equal to '5' for this command
4	CAN ID Filter	CAN identifier to compare incoming message to (MSB first)
4	CAN ID Mask	Bit mask that determines which bits to compare against the filter. 0 = bit will compare true regardless of the corresponding filter bit; 1 = compare bit against the corresponding filter bit (MSB first)
1	Is Extended Filter Flag	0 if only standard identifiers should be accepted; 1 if only extended identifiers should be accepted
1	Is Extended Mask Flag	0 if both standard and extended identifiers should be accepted; 1 if the incoming identifier should be compared to the 'Is Extended Filter Flag'
1	Is Remote Filter Flag	0 if only data frames should be accepted; 1 if only remote frames should be accepted
1	Is Remote Filter Mask	0 if both data and remote frames should be accepted; 1 if the incoming identifier should be compared to the 'Is Remote Filter Flag'

**Returns:** Zero if successful.

### Examples:

The following snippet of PBasic code demonstrates how to set the CAN controller to receive only standard data frames with an ID of "1".

```
'CAN controller commands
CommandFilter      CON 5

'Serial communication parameters
SerialBaudmode CON 6
SerialCts         PIN 7
SerialRfr         PIN 8
SerialTx          PIN 9
SerialRx          PIN 10

'Variables
status VAR Byte

SEROUT SerialTx\SerialCts, SerialBaudmode, [CommandFilter]
SEROUT SerialTx\SerialCts, SerialBaudmode, [0]           'MSB of 32-bit filter id
SEROUT SerialTx\SerialCts, SerialBaudmode, [0]
SEROUT SerialTx\SerialCts, SerialBaudmode, [0]
SEROUT SerialTx\SerialCts, SerialBaudmode, [1]           'LSB of 32-bit filter id
SEROUT SerialTx\SerialCts, SerialBaudmode, [0]           'MSB of 32-bit mask id
SEROUT SerialTx\SerialCts, SerialBaudmode, [0]
SEROUT SerialTx\SerialCts, SerialBaudmode, [$07]
SEROUT SerialTx\SerialCts, SerialBaudmode, [$FF]          'LSB of 32-bit mask id
SEROUT SerialTx\SerialCts, SerialBaudmode, [0]           'isExtendedFilter
```

```
SEROUT SerialTx\SerialCts, SerialBaudmode, [1]          'isExtendedMask
SEROUT SerialTx\SerialCts, SerialBaudmode, [0]          'isRemoteFilter
SEROUT SerialTx\SerialCts, SerialBaudmode, [1]          'isRemoteMask

'Check the return value
SERIN SerialRx\SerialRfr, SerialBaudmode, [status]
```

## Function: Initialize

---

**Description:** Initializes the CAN controller

**Parameters:** 1 byte in length

Field Size (Bytes)	Field Name	Field Description
1	Command Code	Always equal to '6' for this command

**Returns:** Zero if successful.

### Examples:

The following snippet of PBasic code demonstrates how to initialize the CAN controller

```
'CAN controller commands
CommandInitialize CON 6

'Serial communication parameters
SerialBaudmode CON 6
SerialCts      PIN 7
SerialRfr      PIN 8
SerialTx       PIN 9
SerialRx       PIN 10

'Variables
status VAR Byte

SEROUT SerialTx\SerialCts, SerialBaudmode, [CommandInitialize]

'Check the return value
SERIN SerialRx\SerialRfr, SerialBaudmode, [status]
```

## Function: Status

---

**Description:** Returns the status of the CAN controller

**Parameters:** 1 byte in length

Field Size (Bytes)	Field Name	Field Description
1	Command Code	Always equal to '7' for this command

**Returns:** 3 - 16 bytes

Field Size (Bytes)	Field Name	Field Description
1	Controller Status	Status of the CAN controller <ul style="list-style-type: none"><li>• Is the controller ready to transmit</li><li>• Has a message has been received</li><li>• Are there controller errors (see CAN Controller Status below)</li></ul>
1	Controller Errors	Controller error flags (see CAN Controller Errors below)
0 – 13	Received CAN Message	CAN message stored using the minimum number of bytes
1	Call Status Code	Zero if successful

## Examples:

The following snippet of PBasic code demonstrates how to request and receive the status of the CAN controller. Note the constants used to mask out the individual bits in the controller status and controller error fields.

```
'CAN controller status
ControllerState      CON 7 'see "Controller States" below
ControllerReadyToTransmit CON 8 'set IF the controller is ready TO transmit
ControllerMessageReceived CON 16 'set IF a message has been received
ControllerErrorOccured CON 32 'set is an error has occurred

'CAN controller states
ControllerDisabled   CON 0 'the CAN controller is disabled
ControllerErrorActive CON 1 'the CAN controller is "error active"
ControllerErrorWarning CON 2 'still "error active" but error counters > 96
ControllerErrorPassive CON 3 'the CAN controller is "error passive"
ControllerBusOff     CON 4 'the CAN controller is "bus off"

'CAN controller errors
ControllerReceiveOverrun CON 1 'the receive buffer has been overrun
ControllerBitError      CON 2 'a Bit could NOT be transmitted correctly
ControllerFormError     CON 4 'the received CAN message was malformed
ControllerStuffError    CON 8 'expected a stuff Bit
ControllerCrcError      CON 16 'the CRC checksum is wrong
ControllerAckError      CON 32 'the transmitted message was NOT acknowledged

'CAN controller commands
CommandStatus         CON 7

'Serial communication parameters
SerialBaudmode CON 6
SerialCts      PIN 7
SerialRfr      PIN 8
SerialTx       PIN 9
SerialRx       PIN 10

'Variables
```

```

canMessage          VAR Byte(13)
controllerStatus    VAR Byte
controllerErrors     VAR Byte
index               VAR Byte
length              VAR Nib
status              VAR Byte

'Send the status command and receive the controller status and any error codes
SEROUT SerialTx\SerialCts, SerialBaudmode, [CommandStatus]
SERIN  SerialRx\SerialRfr, SerialBaudmode, [controllerStatus]
SERIN  SerialRx\SerialRfr, SerialBaudmode, [controllerErrors]

'Did we receive a message?
IF(( controllerStatus & ControllerMessageReceived ) = 0 ) THEN End_Of_Status

'Retreive the frameInfo and determine the size of the identifier
SERIN SerialRx\SerialRfr, SerialBaudmode, [canMessage(0)] 'Frameinfo
IF( canMessage(0) & FrameInfoExtended ) THEN Rx_Extended_Id

'Retrieve a standard identifier
Rx_Standard_Id:
  SERIN SerialRx\SerialRfr, SerialBaudmode, [canMessage(1)] ' MSB of Std Id
  SERIN SerialRx\SerialRfr, SerialBaudmode, [canMessage(2)] ' LSB of Std Id
  index = 3
  GOTO Rx_Message_Data

'Retrieve an extended identifier
Rx_Extended_Id:
  SERIN SerialRx\SerialRfr, SerialBaudmode, [canMessage(1)] ' MSB of Ext Id
  SERIN SerialRx\SerialRfr, SerialBaudmode, [canMessage(2)]
  SERIN SerialRx\SerialRfr, SerialBaudmode, [canMessage(3)]
  SERIN SerialRx\SerialRfr, SerialBaudmode, [canMessage(4)] ' LSB of Ext Id
  index = 5

'Retrieve the data if it exists
Rx_Message_Data:
  IF( canMessage(0) & FrameInfoRemote ) THEN Can_Return
  length = ( index - 1 + ( canMessage(0) & FrameInfoDataLength ))
  FOR index = index TO length
    SERIN SerialRx\SerialRfr, SerialBaudmode, [canMessage(index)]
  NEXT

'Check the return value
End_Of_Status:
SERIN SerialRx\SerialRfr, SerialBaudmode, [status]

```

## Function: Transmit

---

**Description:** Transmits a CAN message

**Parameters:** 4-14 bytes in length

Field Size (Bytes)	Field Name	Field Description
1	Command Code	Always equal to '8' for this command
0 – 13	Received CAN Message	CAN message stored using the minimum number of bytes

**Returns:** Zero if successful.

### Examples:

The following snippet of PBasic code demonstrates how to transmit a CAN message.

```
'CAN controller commands
CommandTransmit    CON 8

'Serial communication parameters
SerialBaudmode CON 6
SerialCts          PIN 7
SerialRfr          PIN 8
SerialTx           PIN 9
SerialRx           PIN 10

'Variables
status VAR Byte

SEROUT SerialTx\SerialCts, SerialBaudmode, [CommandTransmit]
SEROUT SerialTx\SerialCts, SerialBaudmode, [canMessage(0)] ' Frameinfo
IF( canMessage(0) & FrameInfoExtended ) THEN Tx_Extended_Id

Tx_Standard_Id:
  SEROUT SerialTx\SerialCts, SerialBaudmode, [canMessage(1)] ' MSB of Standard Id
  SEROUT SerialTx\SerialCts, SerialBaudmode, [canMessage(2)] ' LSB of Standard Id
  index = 3
  GOTO Tx_Message_Data

Tx_Extended_Id:
  SEROUT SerialTx\SerialCts, SerialBaudmode, [canMessage(1)] ' MSB of Extended Id
  SEROUT SerialTx\SerialCts, SerialBaudmode, [canMessage(2)]
  SEROUT SerialTx\SerialCts, SerialBaudmode, [canMessage(3)]
  SEROUT SerialTx\SerialCts, SerialBaudmode, [canMessage(4)] ' LSB of Extended Id
  index = 5

Tx_Message_Data:
  IF( canMessage(0) & FrameInfoRemote ) THEN Can_Return
  length = ( index - 1 + ( canMessage(0) & FrameInfoDataLength ))
  FOR index = index TO length
    SEROUT SerialTx\SerialCts, SerialBaudmode, [canMessage(index)]
  NEXT

SERIN SerialRx\SerialRfr, SerialBaudmode, [status]
```

## Function: Set Bit Rate

---

**Description:** Sets the CAN controller's bit rate (see the 'Bit Timing' command as an alternative)

**Parameters:** 3 bytes in length

Field Size (Bytes)	Field Name	Field Description
1	Command Code	Always equal to '9' for this command
2	CAN Bit Rate	CAN bit rate in 1000's of bits/second. (Must be a value of 10, 25, 50, 125, 250, 500, 800 or 1000)

**Returns:** Zero if successful.

### Examples:

The following snippet of PBasic code demonstrates how to set the CAN bit rate to 125 kbits/sec.

```
'CAN controller commands
CommandBitRate    CON 9

'Serial communication parameters
SerialBaudmode CON 6
SerialCts         PIN 7
SerialRfr         PIN 8
SerialTx          PIN 9
SerialRx          PIN 10

'CAN bit rate settings
BitRate CON 125

'Variables
status VAR Byte

SEROUT SerialTx\SerialCts, SerialBaudmode, [CommandBitRate]
SEROUT SerialTx\SerialCts, SerialBaudmode, [(BitRate & $FF00)]
SEROUT SerialTx\SerialCts, SerialBaudmode, [(BitRate & $FF)]

SERIN SerialRx\SerialRfr, SerialBaudmode, [status]
```

## Function: Receive All

---

**Description:** Sets the CAN controller's acceptance filter to receive all messages (use as an alternative to the "Set Filter" function).

**Parameters:** 1 byte in length

Field Size (Bytes)	Field Name	Field Description
1	Command Code	Always equal to '10' for this command

**Returns:** Zero if successful.

### Examples:

The following snippet of PBasic code demonstrates how to set the CAN acceptance filter to receive all messages.

```
'CAN controller commands
CommandReceiveAll CON 10

'Serial communication parameters
SerialBaudmode CON 6
SerialCts      PIN 7
SerialRfr      PIN 8
SerialTx       PIN 9
SerialRx       PIN 10

'Variables
status VAR Byte

SEROUT SerialTx\SerialCts, SerialBaudmode, [CommandReceiveAll]

SERIN  SerialRx\SerialRfr, SerialBaudmode, [status]
```

## Function: Disable LEDs

---

**Description:** Disables the CAN controller's status LEDs to conserve power

**Parameters:** 1 byte in length

Field Size (Bytes)	Field Name	Field Description
1	Command Code	Always equal to '101' for this command

**Returns:** Zero if successful.

### Examples:

The following snippet of PBasic code demonstrates how to disable the status LEDs on the MCI-100P.

```
'CAN controller commands
CommandDisableLed CON 101

'Serial communication parameters
SerialBaudmode CON 6
SerialCts      PIN 7
SerialRfr      PIN 8
SerialTx       PIN 9
SerialRx       PIN 10

'Variables
status VAR Byte

SEROUT SerialTx\SerialCts, SerialBaudmode, [CommandDisableLed]

'Check the return value
SERIN SerialRx\SerialRfr, SerialBaudmode, [status]
```

## Electrical Characteristics<sup>2</sup>

### *Absolute Maximum Ratings<sup>3</sup>*

Operating Temperature	-40°C to 85°C
Storage Temperature	-65°C to 150°C
Voltage on Tx, Rx, RFR, and CTS pins with respect to ground	-0.5 to $V_{cc} + 0.2V$
Voltage on CAN_H and CAN_L pins with respect to ground	-36V to 36V
Voltage on Vcc with respect to ground	-0.5 V to 6.0 V
DC current on Tx, Rx, RFR and CTS pins	10 mA max

### *DC Characteristics*

Symbol	Parameter	Minimum	Typical	Maximum	Units	Note
Vcc	Supply Voltage	4.38	5.00	6.00	V	

---

<sup>2</sup> Electrical Characteristics for this product have not yet been finalized. Please consider all values listed herein as preliminary and non-contractual.

<sup>3</sup> Stresses beyond those listed under “Absolute Maximum Ratings” may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or other conditions beyond those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

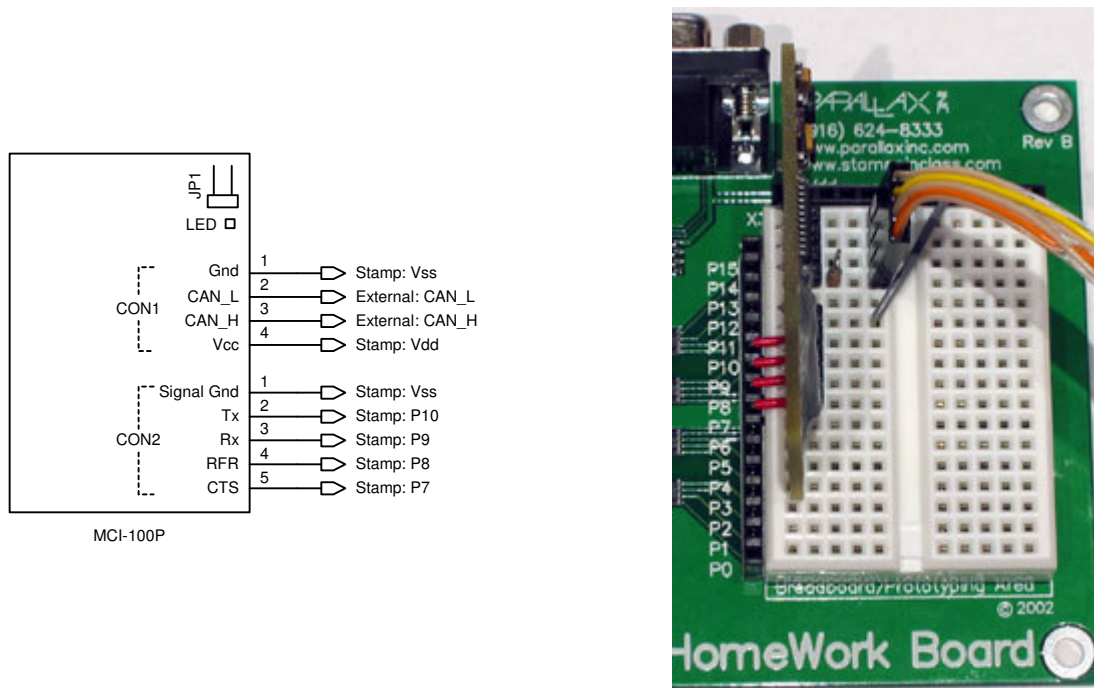
## Appendix A: Basic CAN API Example

The purpose of this experiment is to show what it takes for the Stamp to be CAN aware. The PBasic software configures the MCI-100P, writes a copy of every message on the CAN bus to the Stamp's debug terminal and echoes the message on the CAN bus, if configured to do so<sup>4</sup>.

The MCI-100P presents an API we call the Basic CAN Controller. This API is our wrapper around a Basic CAN Controller (see The Basic CAN Controller API for more information). The PBasic example contains all the constants, variables and code needed to make full use of the Basic CAN Controller API. It is possible to reduce the amount of code by using only a subset of the Basic CAN Controller API's features. The example in Appendix B shows how this is done.

### Hardware Setup

The diagram and picture in Figure 9 show how to connect the MCI-100P to the BASIC Stamp HomeWork Board. In this example the MCI-100P is powered off of the BASIC Stamp's power regulator. Also, note the terminating resistor located between the CAN bus cable and the MCI-100.



**Figure 5:** The MCI-100P connected to the BASIC Stamp HomeWork Board

<sup>4</sup> Warning: Do not run this example on a CAN bus where there is the potential for damage or injury. Altering the state of a working system can cause unexpected results.

## **Software Setup**

It takes a few steps to get the Basic CAN Controller API enabled. Once enabled, the Stamp is able to transmit and receive CAN messages. The PBasic Code for this example can be found on the Machine Bus web site at the URL <http://www.machineBus.com/downloads/mci100p.zip>.

### **API Initialization**

Perform the following steps to enable the Basic CAN Controller API:

- Step 1)** Establish the serial communication rate
- Step 2)** Verify that you're in the command state
- Step 3)** Initialize the Basic CAN controller
- Step 4)** Set the CAN bit rate or individual CAN bit timings (see 'Bit Rate' or 'Bit Timings' in The Basic CAN Controller API)
- Step 5)** Set the acceptance filter (see 'Receive All' or 'Filter' in The Basic CAN Controller API)
- Step 6)** Enable the CAN controller
- Step 7)** Enter processing loop

These steps are clearly labeled in the PBasic example. See [The Basic CAN Controller API](#) for a full description of the boot process.

### **Transmitting and Receiving CAN Messages**

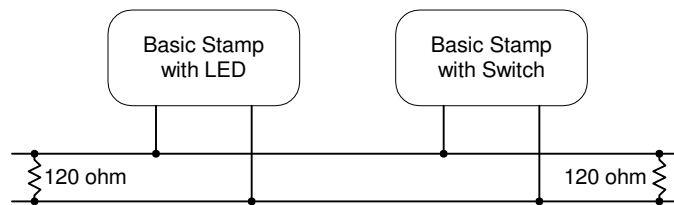
In order to transmit and receive, you must continually poll the status of the CAN Controller. The status tells you the bus state of the controller, if any errors have occurred, if a message has been received and if the controller is ready to transmit. See [The Basic CAN Controller API](#) for a full description.

## Appendix B: Blink

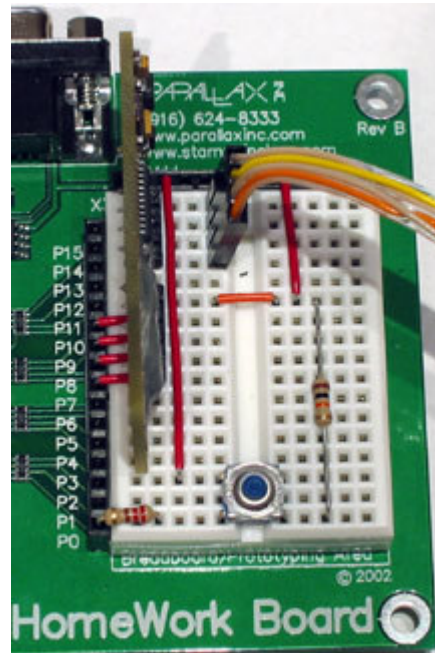
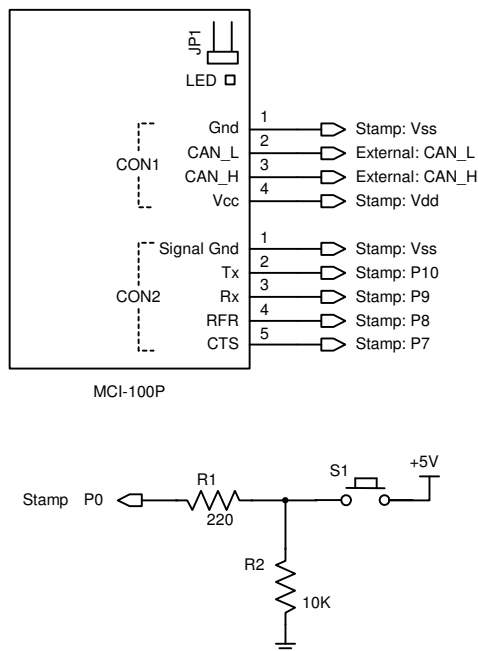
The purpose of this experiment is to demonstrate CAN communication between two Stamps. We'll add a switch to one Stamp, and a bi-color LED to the other. The state of the switch will control the color of the LED.

### Hardware Setup

Two boards will have to be setup for this experiment. Diagrams and pictures of the completed boards are shown in figures 11 and 12. Notice the 120 ohm terminating resistor between CAN\_L and CAN\_H. It is always a good idea to properly terminate the bus, even though it may not be necessary with these short bus lengths. In this experiment the MCI-100P receives its power from the Basic Stamp's power regulator.

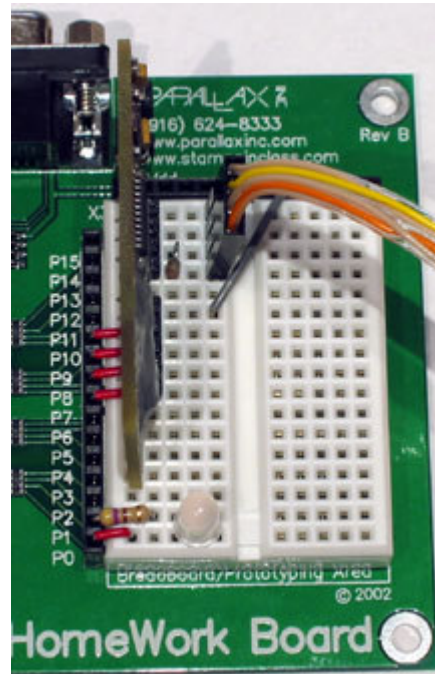
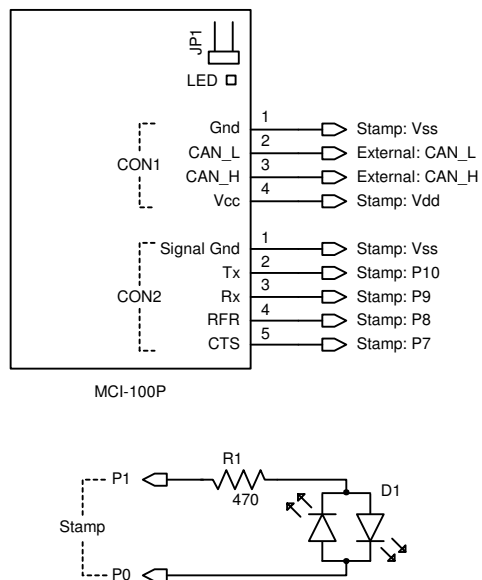


**Figure 6:** Both Stamp devices are connected via the CAN bus



**Figure 7:** Schematic and a picture of how to connect a switch and the Stamp/CAN Interface to the Basic Stamp

Update picture and diagram



**Figure 8:** Schematic and a picture of how to connect a bi-color LED and the Stamp/CAN Interface to the Basic Stamp

## Running the Software

The PBasic Code for this example can be found on the Machine Bus web site at the URL <http://www.machineBus.com/downloads/mci100p.zip>.

**Step 1)** Execute the PBasic file `canSwitch.bs2` on the board with the switch.

**Step 2)** Execute the PBasic file `canLed.bs2` on the board with the LED.

The color of the LED should alternate between red and green each time the switch is pressed and released.