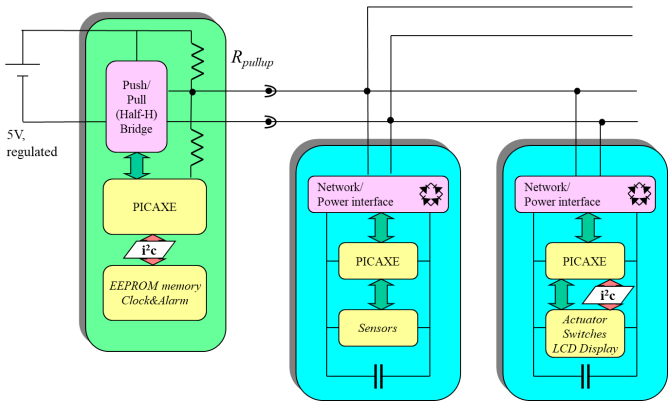
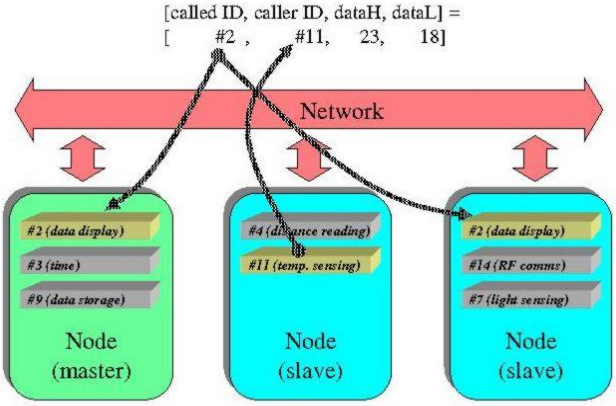
PICAXE™ "SerialPower I" Network V4.0

For PICAXE M2/X2 type microcontrollers

Combining power delivery and
bi-directional communications between intelligent nodes
using just two interchangeable wires





JURJEN KRANENBORG

Document version 4.0
(October 2018)

Trademarks:

PICAXE™ is a trademark of Revolution Education, Ltd., Great Britain
LEGO MindStorms™ is a trademark of The LEGO Group, Denmark.

Disclaimers:

The author of this document cannot accept any responsibility in any way for any application of the concepts presented here.

The author of this document is not related in any way to any component producer or other company referred to in this document.

Contents

PICAXE™ "SerialPower" Network

Jurjen Kranenborg
http://www.kranenborg.org/jurjen-cv

1. Summary and motivation

The PICAXE "SerialPower" Network allows combined power delivery and bi-directional data transfer between processes on a number of intelligent nodes over just two, interchangeable wires possibly spanning tens of meters. The network consists of a master node and (several) slave nodes, each node containing a PICAXE microcontroller. The master node manages for power delivery and the provision in a regular fashion for timeslots during which processes on a slave node can exchange information with processes on other nodes using a fixed data frame. Slave node processes have functional behavior (which master node processes may provide for as well). Slave nodes need just a capacitor for energy storage and buffering. Power-demanding nodes with local power sources can be added as well without any change. Simple diode-mixing networks with separate power and communication lines can be used with the same software protocol.

The network has the following characteristics:

- Multi-drop bi-directional network (half-duplex), allowing information transfer between any combinations of nodes. Nodes that only "listen" (i.e. do not have sending processes) may be implemented with reduced hardware and software complexity.
- The concept of communicating processes is used (each process has a unique identifier), allowing abstraction from physical nodes and thereby flexible distribution of functionality over different nodes. A node may implement several processes, and the same process may run concurrently on several nodes with local modifications. As a result extremely flexible nodes can be designed that can be reconfigured via the network. Data is transferred over the network using a data-frame containing the caller process identifier, the calling process information and a number of data bytes. The network protocol completely abstracts from the underlying network hardware, allowing the application of very simple "diode mixing" network hardware with separate power and communication lines as well.
- Data transfer and power delivery to nodes over two simple wires (no additional GND or handshaking lines!), to which an unlimited number of nodes can be connected in any fashion. Consequently, the network complexity and software complexity does not necessarily scale with the number of nodes.
- The network connections are non-polarized, i.e. any node can be connected in any fashion to an existing two-wire network using two identical and interchangeable wires.
- The master node provides for power (delivered by default). Slave nodes have a local capacitor for uninterrupted operation when the network is pulled low during communications or interrupts.

- All nodes may contain a PICAXE M2 or X2 microcontroller to implement the network (requiring only a small part of the available program memory) as well as functional behavior (for example for sensing, switching or signaling purposes).
- The network protocol implementation is strongly based on interrupt handling, allowing the nodes to concentrate on functional operation for most of the time. When interrupted, all nodes read all data frames on the network, but quickly return to normal operation if they do not implement one of the processes relevant to the particular data frame.
- The network stack is implemented with an "intelligent" master node that roams for sending slave processes after network power-up, and auto-registering slave nodes that respond with the IDs of their sending processes. Thus, the master node becomes completely application independent (avoiding reprogramming) and focuses on timeslot provision in an efficient manner. Additionally, the master node provides for processes to add extra timeslots for other slave processes that wish to send, allowing "Plug & Play" slave nodes to be added on the fly.
- Through the master node the microcontroller frequency of the slave nodes may be set at suitable values (between 256 KHz and 32 MHz) to balance requirements for speed, power consumption etc. .
- Simple and cheap circuitry for network electrical implementation using solid state components only (no analog components like transformers used).
- The network concept can be extended in many ways, like adding CRC checking, prioritized node timeslot assignment, interrupting slave nodes, longer data frames, adding node specific processes for energy consumption regulation etc.
- Energy consuming nodes that need their own energy source can be used without any modifications to the nodes
- Minimum setup of two nodes is possible in which the master node implements functional processes as well (at the penalty of some network bandwidth reduction).
- A full slave network stack on a PICAXE-08M2 still leaves ample room for applications.
- Although extremely well suited to the PICAXE microcontroller, other types of microcontroller can be applied in network nodes as well (in particular those that can operate below 5V).

The idea of developing a purely two-wire bi-directional network came to my mind after I had read about home-made sensors for the LEGO™ original MindStorms robotics package. The sensors used for this system have a backup capacitor. Prior to reading they are powered from the main controlling module for a short while, subsequently an analog value related to the sensor is read. The simplicity of this interface has motivated a number of people to develop their own sensors; initially I had thought of making a PICAXE-based design for this type of interface. Since then LEGO has upgraded the sensor interface to a digital one.

The analog interface has a number of drawbacks. First of all, only one analog value can be read at any time. Second, in case an accurate sensor measurement is needed, a more complex interface at the sensor side is needed. Furthermore, the analog interface does not

allow any form of addressing of multiple sensors (although some hardware hacks have been developed which sometimes allow a few more simple sensors to be used).

I quickly realized that a completely digital interface that at the same time provides for continuous power would be much more flexible as well as support more power-hungry applications. The current document is the result of on-and-off work for several years on developing such an interface. I decided to develop my own network "standard" in order to learn as much as possible, although it eventually morphed into something that is quite alike official standards, in particular LINBUS. I also learned much about the analog aspects of digital networks; although we are creating a digital network, the fact that we are dealing with relatively low voltages, back-up capacitors and MosFET gate threshold voltages just below the main voltage leads naturally to issues of analog nature.

I hope that the work published in this document inspires people to both develop applications as well as investigate improvements on the presented network concept. I believe that a network concept like the one presented here can lead to applications that are more powerful as well as much cheaper than those based on RF communications. A particular attractive feature is that very cheap but also very functional nodes can be built with the exceptionally powerful but small PICAXE-08M2 microcontroller. The integration of a 2D network as part of (modules of) a 3D-printed design clears the path for applications previously unheard of.

I wish to remark that the work presented here is a tribute not only to the PICAXE concept but also to the PICAXE forum [2]. I have been inspired by many discussion threads, and readers will probably recognize many elements. For example, the network messaging approach is strongly based on "hippy's" excellent treatment of serial interrupts [3]. Furthermore "wilf_nv" has contributed greatly with a much simplified electrical design for the master node. "Puddlehaven" has contributed to the interpretation of the network operation through his SerialPower-based application. But also other members have contributed. It is this combination of excellent hardware, support and active user base that makes the PICAXE concept ideal for designing systems.

All documentation as well as software implementations (including future revisions) are available for download from the authors home site on electronics designs:
http://www.kranenborg.org/electronics

2. Application areas

The “SerialPower” Network is particularly useful for short to medium range networks (tens of meters) over which low-intensity information exchange is required and network electrical stability is good. Application areas include:

Home monitoring systems: Examples of useful applications include energy monitoring (temperature and light), climate regulation, safety monitoring and local traffic monitoring.

Weather monitoring: outdoor intelligent sensors can be placed at different locations to report on meteorological conditions.

Information presentation: Information can be presented easily at several locations, possibly dependant on local context.

Robot design: Multiple sensors and actuators and switches can be managed with reduced wiring effort in modular designs. Although the network is not designed for operations that require large power consumption suddenly (starting up motors for example), the backup capacitor in slave nodes can be replaced directly by a local power source, while communication occurs over the network as usual.

Sensor fusion networks: Powerful high-level sensors can be built that integrate many physical sensor types in an intelligent node.

Beacon networks: A networked set of beacons can be developed that supports IR or ultrasound bi-directional communications with local modules, allowing the latter to perform out-of-sight communications. Also various applications in robotics can be thought of, like for example position determination.

Model railway automation: Trafficking light signals and large amounts of passage detection sensors and/or switches can be managed and powered without complex wiring in tough-to-reach environments where battery replacement is difficult. Furthermore, the network concept allows easy extension with new nodes.

PART 1: GENERAL OVERVIEW

3. Operation and technical concepts

This part gives a short overview of the functioning of the "SerialPower" network, starting with the network hardware. A thorough overview of all concepts and their relations is given in Part 2 of this document ("Architecture"), which starts with the high-level logical concepts instead.

Hardware and logical views

Figure 1 shows a hardware-oriented impression of the network. It shows a master node that manages for power delivery and timeslot provision to all nodes. The slave nodes have a backup capacitor for use during timeslots and interrupts on the network.

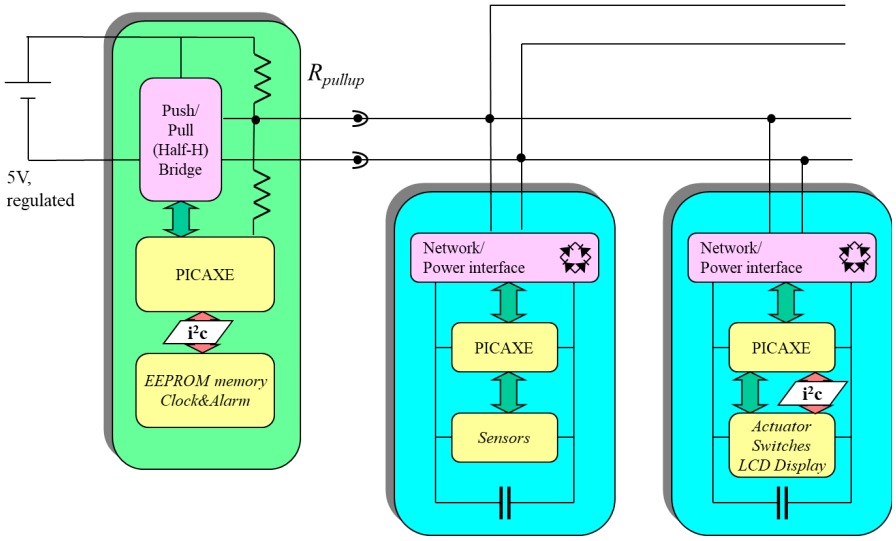


Figure 1: Hardware view on the "SerialPower" network concept

The network consists of a pair of simple wires. In its most "pure" form the slave network/power interfaces of a "SerialPower" network consist of a four-diode DC rectifier bridge, allowing the wires to be connected to the nodes in a non-polarized, i.e. interchangeable fashion. Slave nodes can be added to the network in any fashion and any number.

Now imagine that the hardware setup from Figure 1 is used as a demonstrator of SerialPower. The logical view on this system as depicted in Figure 2 below completely abstracts from the underlying hardware, as the major concepts are processes and messages between them.

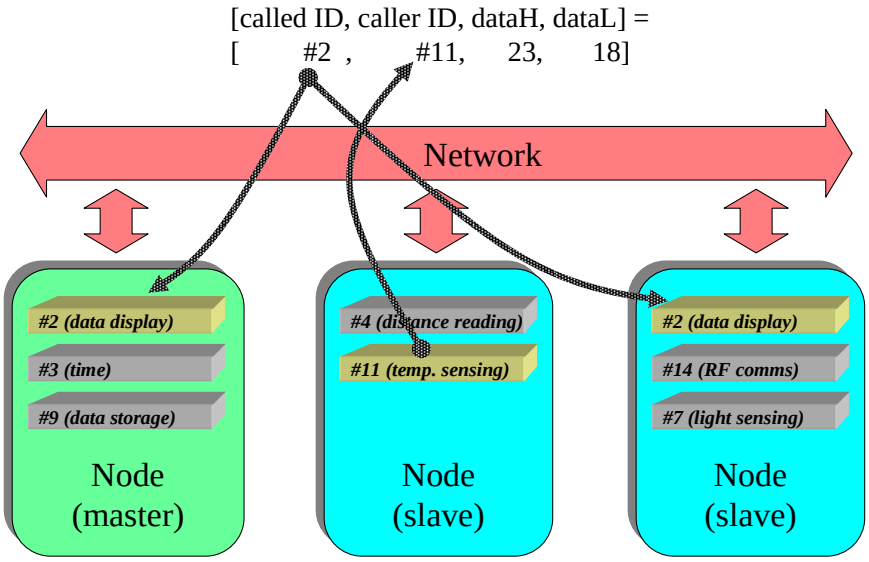


Figure 2: Logical view on the "SerialPower" network concept

A process can communicate with any other process using a set of standardized messages. In SerialPower, these messages contain the ID of the process that is addressed, the calling process ID, and some data bytes. On this level of abstraction the nodes themselves are not visible in any way; a process may be distributed over several nodes, and on a node several processes may reside. This implies that slave-to-slave is the standard way of communication. Both aspects are depicted in the figure, where a temperature sensing process (with ID 11 that happens to reside on some slave node) sends information to a displaying process (with ID 2, this process happens to be distributed over two nodes, namely both the master node and a slave node).

Since the logical view completely abstracts from the network hardware, the software that implements the processes and their communication in a "SerialPower" network can be used with very small adaptations to any other, simpler type of interrupt-driven networking hardware with separate power lines. Such a simple solution is presented in Figure 3 and dealt with further in Chapter 7.3.

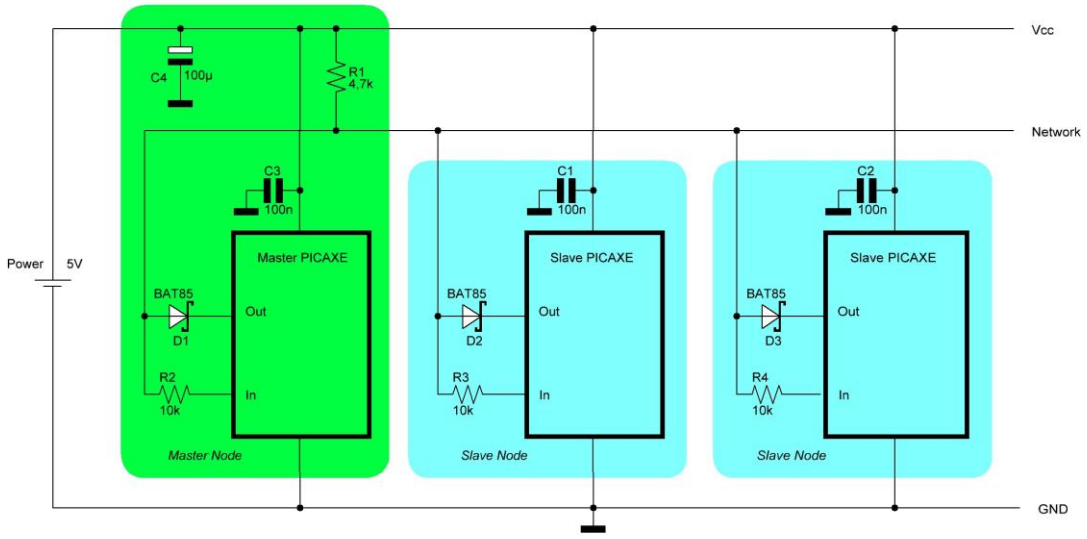


Figure 3: Simple network design with separate provision for power and serial communication.

Master and slave node networking hardware description

One "master" node provides for power as well as "timeslots" during which sub-programs ("processes") on one or more "slave" nodes can use the bus for a predefined time interval. Figure 4 shows the hardware principles of a network consisting of a master node (left) and just one slave node (right).

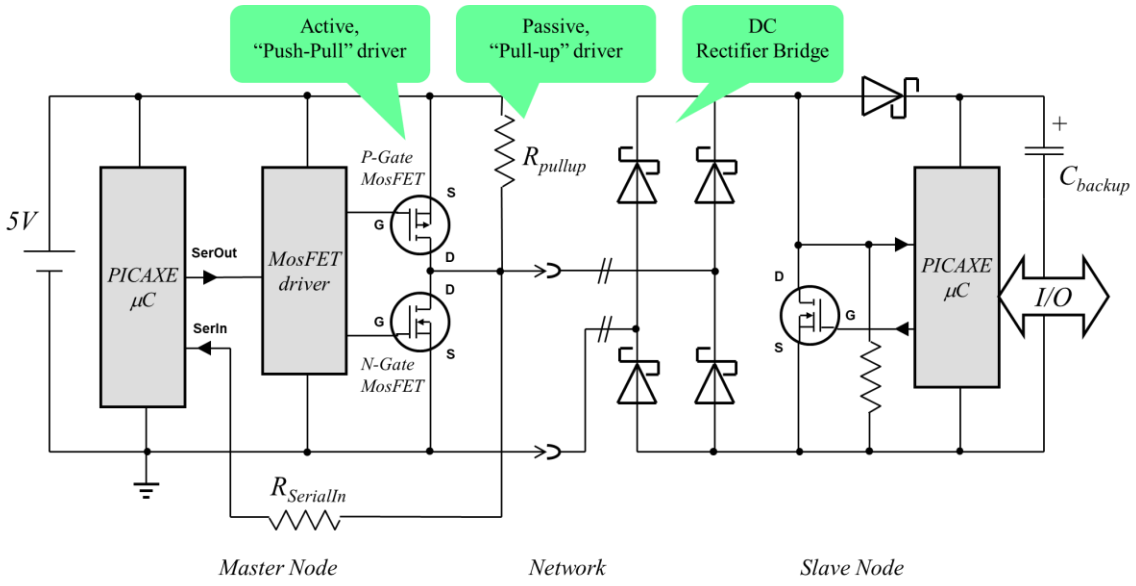


Figure 4: Main interface elements of the network. The master node is on the left.

The master node has an active driver consisting of two MosFETS. This driver is used to power the network or to transmit master node messages to the network. When the network is to be used by slave nodes the active driver is 3-stated by the master and the passive pull-up resistor keeps the network at high level for a certain period. Thus a "timeslot" is created for use by the slave nodes. Subsequently a slave node can send zero bits by locally pulling the network low. If the network is not pulled low, the passive pull-up keeps the network at a logical high level. At the slave node a DC rectifier bridge based on Schottky diodes allows for combined power delivery and information transfer from the slave controller. Note that a timeslot is only allowed to last for a short period of time, as the slave nodes are then dependent on their local backup capacitors for power provision.

Figure 5 shows the current flow in the network when the master controls the network during network powering or during the transmission of a master-initiated serial message transmission. Note that the nodes are powered during transmission of 1-bits.

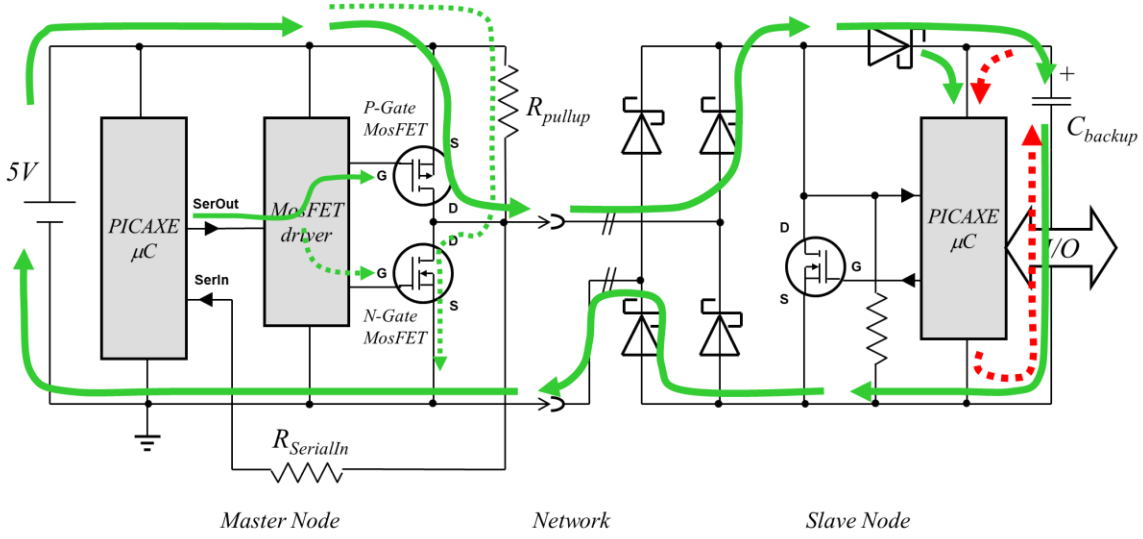


Figure 5: Current flow when the active, push-pull driver is used. Straight arrows correspond with a high level network state; dotted arrows correspond with a low level network state.

Figure 6 shows the current flow during a timeslot when a slave communicates. The network can be pulled low by the slave to send a LOW logical level ("0"-bit). In that case the backup capacitor is the power source for the slave node. A HIGH logical level ("1"-bit) is created by closing the slave node MosFET.

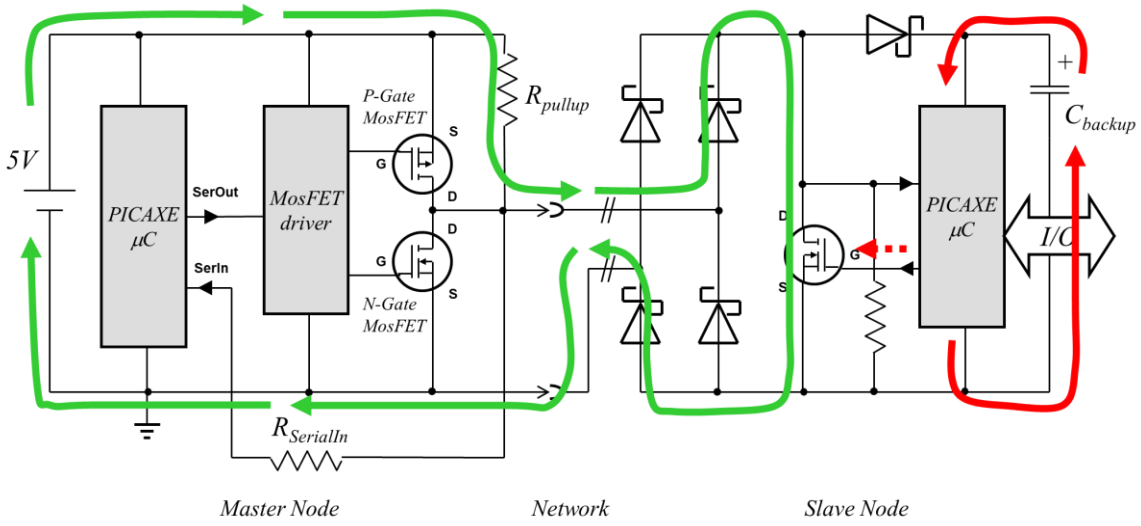


Figure 6: Current flow during a communication timeslot when the passive pull-up driver is used and the slave pulls the network low, either for sending an interrupt or creating a logical low level. A green arrows shows current supplied by the master, a red color denotes current provided by the slave back-up capacitor.

Note that the master node PICAXE can read network messages through $R_{serialIn}$.

Communication

The network physically consists of only two lines without any additional handshaking lines. Therefore communication is managed based on the following principles:

- The network is at high logical level by default to allow power delivery to all slave nodes.
- All communication is initiated and timed by the master node trough master commands (i.e. master node messages).
- All messages are transferred as a result of an interrupt, caused by a master or a slave pulling the network low.
- A message always has the same format.
- The master node uses special messages (e.g. `availableTimeSlot`) to indicate that a network slot is available for a specific process during which the latter may send a message itself (directly after the master message). This special message also indicates – through a process ID - which process may use the timeslot to send a message, thus avoiding network message collisions.

The list below shows all possible cases for messaging sequences as embedded in bus states:

1. HIGH, INT, MasterMessage, HIGH
2. HIGH, INT, MasterMessage, HIGH(weak pullup), INT, SlaveMessage, HIGH
3. HIGH, INT, MasterMessage, HIGH(weak pullup), HIGH

In this list "HIGH" means that the network level is driven high through the active push-pull driver, i.e. power is supplied to the nodes. "HIGH (weak pull-up)" means that the active driver is 3-stated and the network level is held high through the master pull-up resistor, INT implies that the bus is shorted (either by the master or the slave) to trigger a network interrupt and indicate to all nodes that a message is underway.

1. In Case 1 the master sends a message and then returns to power provision again.
2. In Case 2 the sequence starts in the same manner, but now the master message is a special command (`availableTimeSlot`) indicating that a certain process on some slave node is granted a timeslot to send a message in the same way as the master did. Consequently the master creates a timeslot by applying the passive pull-up driver to the network. If the particular process has a message to send, it will do so by quickly pulling the network low and sending its message. Afterwards, the master node reactivates the active driver to restore power provision.
3. If a process is granted a timeslot but does not use it then Case 3 applies, i.e. after a defined period of time the master replaces the passive driver with the active power driver.

Node Configuration

The actual distribution of processes on nodes is very flexible, and can range between the following options:

- Thin master, large slave nodes: In this scenario there are a large number of approximately equally important processes distributed over the various slave nodes that communicate intensively with each other, and the primary goal of the master node is to provide for a fair distribution of timeslots amongst the communicating processes.
- Large master, thin slave nodes: This would be appropriate if the slaves act as peripherals to the master node, and there is a central process on the master node that communicates to the slave processes (while the slaves require little interaction themselves).
- Thin "intelligent" master node for timeslot provision, one slave node as master for functional processes, plus remaining thin slave nodes: Since the "intelligent" master node can be ordered to provide or remove additional timeslots, the "functional master" node can be a slave node itself which does not need to

continuously manage timeslots anymore. Furhtermore the timeslot master node is then completely application independent.

Which distribution of processes amongst the modes is best depends much on the application and the resources that nodes can provide to processes.

Simplified polarized network interface for slave nodes

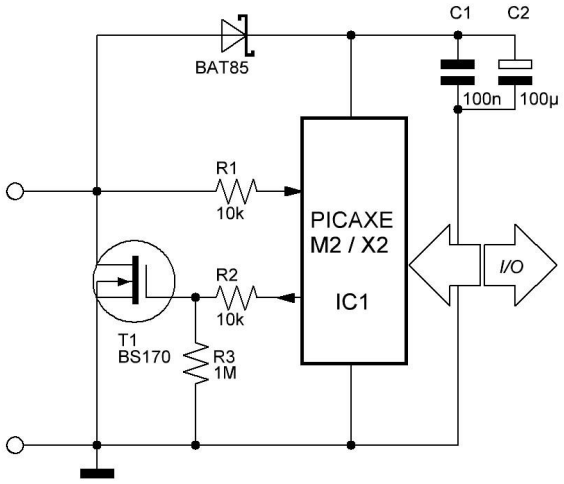


Figure 7: Polarized slave node connection

In case the polarization of the power/data network is known a much simpler interface for slave nodes can be used, as show in Figure 7. Section 7.1 deals further with this hardware simplification.

Intelligent, network roaming master node / auto-registering slave nodes

In order to avoid reprogramming of the master node for each application and to allow slave node processes to add and remove timeslots for other process on the fly, an intelligent, network roaming master node / auto-registering slave node software stack is presented in Chapter 9. This leads to a completely application-independent master node and results in much easier and flexible application development. It is the recommended software stack to be used, and the programming guide and example routines in Chapters 10-11 are based on this concept. Both master and slave node network stacks fit easily within a PICAXE-08M2.

Plug & Play network slave nodes (UNTESTED)

An intelligent master node that itself can roam the network for sending processes and collects their IDs allows the use of Plug & Play type nodes that can be connected to the network on the fly. This special node type is shown in Figure 8 below and is dealt with in Section 7.4.

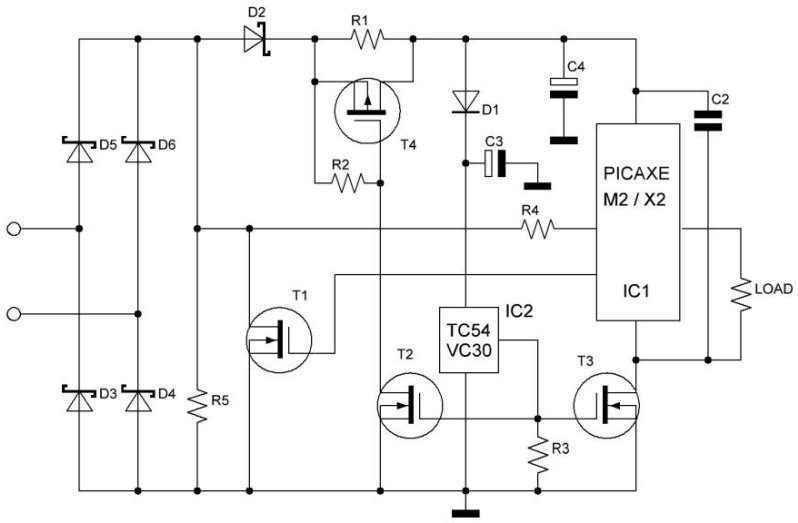


Figure 8: Hardware for a Plug & Play type slave node

PART 2: ARCHITECTURE

4. Network logical concepts

4.1 Logical view I: Nodes, Processes and Message Frames

Central to the network architecture are the following concepts:

- Processes: perform some useful activity
- Nodes: contain one or more processes, providing them with resources
- Network: allows processes to exchange information via data transfer
- Message Frame: Data format by which processes exchange information

Figure 1 shows how these concepts are related. A node may have several processes running; each process is uniquely identified through a number (byte). A process may be distributed over multiple nodes (in Figure 1 the data display process with ID #2 is distributed over two nodes, one node may display a message using a LCD, another node may display the same message using a 7-segment display).

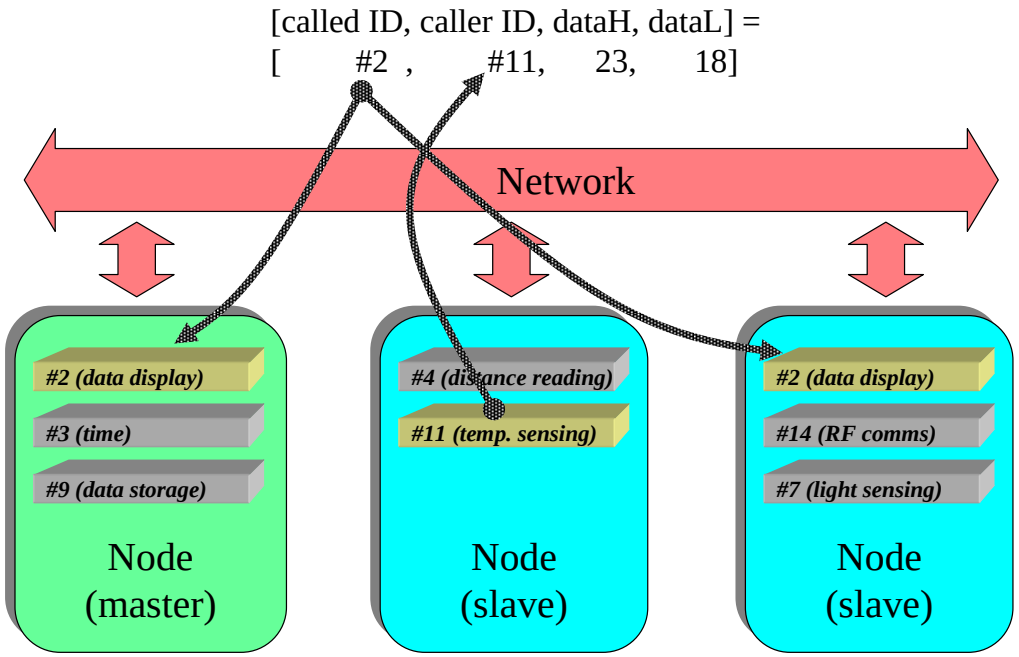


Figure 1: Components of the network

All processes communicate via a multi-drop network by placing a message frame on the bus in an appropriate timeslot. Figure 2 depicts this message frame. It contains the process that is to be addressed, information from the caller process (i.e. the process that put the frame on the network, for example it could be the sending process ID), and a few data bytes.

These message frames are handled by the network stacks present on each node and passed to the process that is addressed. From the message point of view it is unimportant on which physical node the addressed process is located, as all nodes (including the master) read the frame in an equal manner.

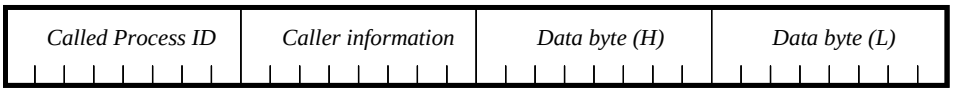


Figure 2: Message Frame format

Figure 1 also shows that there are two different types of node: a master node and one or more slave nodes. One function of the master node is to provide for timeslots during which data frames can be put on the network. The next sections deal with this aspect in more detail. From a message point of view the nodes are perfectly equal, and the message formats used by the master and slaves are equal as well.

4.2 Logical view II: Protocol and bus timing

We are dealing with a pure two-wire bus on which both power delivery and message exchange have to take place while handshake signal lines are absent. Consequently, we rely on well-defined bus state sequencing and timing definitions, with the master node defining the latter.

Figure 3 shows the network bus timing diagram for the most complete (and most complicated) case: a bus transaction in which a slave node process is allowed to put a message frame on the network bus. The timing interval names are those that are used in the network stack software.

The description of the bus timing in Figure 3 starts on the left side by assuming that the bus is in a high state (meaning that the voltage difference between the lines is sufficient to define a High logical level). In this state power is supplied over the bus to the slave nodes by the master node through the high side of a push-pull (or half-H) MosFET bridge.

At instance A the bus is brought low by the master node (the high-side of the bridge is closed while the low side is opened, effectively causing a short-cut of the network lines. The slave nodes now have to rely on their local capacitor for their energy needs). This signals that a message frame is to be awaited by all other nodes, and an interrupt is generated in the nodes causing them to listen for a serial message. The bus is kept low for a time interval equal to *t_INT*, allowing the slave nodes to interrupt their current program execution and get into listening mode.

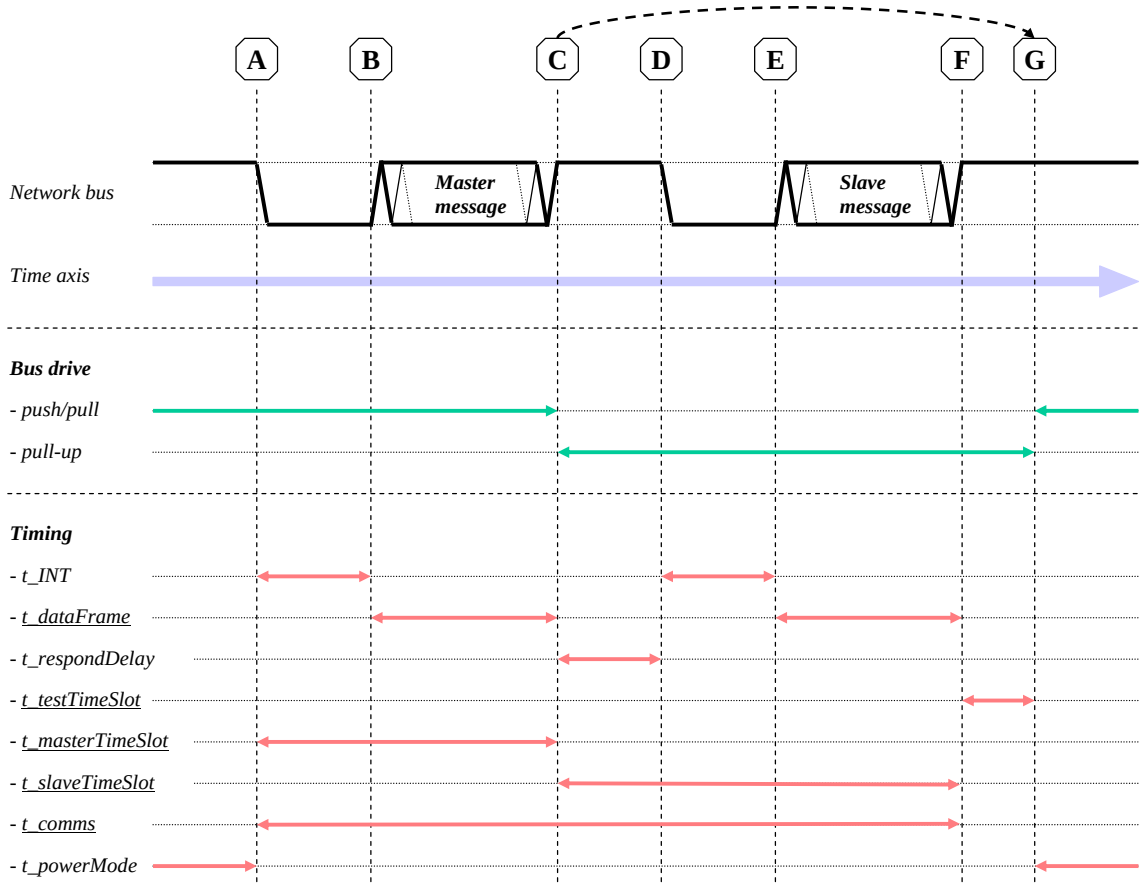


Figure 3: Bus messaging sequences, drive characteristics and timing for a single bus transaction

At instance B the master node sends a message frame on the bus. This frame is read by all slave nodes. The time interval required to send the message is equal to *t_dataFrame* .

Immediately after the completion of the message frame transmit, i.e. at instance C, two situations may apply.

If the message frame is defined not to lead to a response from a process on a slave node, the bus level can be brought high again by the master node for power delivery, and instance G applies. *No process is then allowed to try to take the bus.*

In all other cases at most one process located at no more than one node should be granted the possibility to take control over the network to send a message. To arrange for this, the master node has put a special master command on the network that is called `availableTimeSlot` (see Chapter 9). In this case, at instance C the bus is brought high again by the master node, but now through a passive pull-up resistor (and the push-pull driver is 3-stated). The total amount of time that this condition exists is equal to *t_slaveTimeSlot* + *t_testTimeSlot*.

After some time period *t_respondDelay* a process one some node may take control of the network bus by pulling it low at instance D. Now the same situation appears as at instance A (but with the bus resistive pull-up used instead of the active driver); an interrupt occurs on all nodes (now possibly also including the master node) and subsequently after a period *t_INT* at instance E the message is read. At instance F the message is sent and after a short period t_*testTimeSlot* (potentially useful for bus testing purposes by the master node) the bus is brought high again at instance G by the master node for power delivery.

In the timing diagram of figure 3 the underlined time interval names are maximum durations and are defined by the master node. Consequently slave nodes need to adhere to these definitions.

Note that the way to initiate a message transfer is similar for both master and slave node messages. Also short period of time equal to *t_respondDelay* is available between the two transfers. Consequently, both message types can be handled by the same software routine responsible for handling the interrupt and reading the messages. In fact, from a software point of view there is no difference between the master and slave node message. This approach leaves us with a very compact network stack. For example, a fully functional network stack for bi-directional transfer requires 295 bytes on a PICAXE-08M2 based slave node, leaving ample room for useful applications. In case a slave node has only processes that receive and do not send, the network stack can be reduced even more. Chapter 9 presents a generic implementation of the network stacks for master and slave nodes, as well as a reduced network stack for slave node that implements listening-only processes.

5. Software concepts

The following concepts can be generally found on any node:

Functional processes: These are the sub-programs that do the functional work, like reading sensors, displaying information etc. These processes either run in the "background" (i.e. the program body) and may be interrupted when a message appears on the network, or are started as a result of a received message addressing the particular process. Processes may pass data to the network stack via certain memory variables or registers.

Network stack: This comprises an interrupt routine that reacts on the network input monitoring line to go low, indicating that a message frame is on the way. Furthermore the routine reads the message, stores the message components locally, decodes the called process ID and starts up the referenced (called) process if it is available on the particular node. On a slave node it also may subsequently send a message frame if a process on the node is granted a timeslot on the network by the master and the particular process actually has a message (i.e. data) to send. Other parts of the stack include the definition of

some variables for message passing between a slave process and the network stack in order to create a message frame that the process wants to have transmitted. Finally, some initialization code is included.

Dedicated processes: Dedicated processes may be used to implement parts of the network stack or for example enable processes on nodes. Examples of such processes that are described in the examples of Chapter 9 are:

- `availableTimeSlot`: This process is part of the network stack and should be implemented by all slave nodes that have one or more processes that might want to put data on the network. It is used by the master node to generate a timeslot for the particular process. The associated message frame is:

  [IDavailableTimeSlot, IDcalledProcess, dataByteH, dataByteL]

  By sending this message frame to the network, the master node indicates to some process corresponding with IDcalledProcess that it is allowed to subsequently put a message frame on the network. See sections 9.1 and 9.2 for examples.
- `registerSendingprocess`: This process can be used by slave nodes or slave processes to request timeslots for other processes.

Synchronization routines: A synchronization concept is necessary for two reasons:

1) Execution of time-consuming instructions: The PICAXE has a code interpreter, which implies that code instruction speed is relatively low (order of a millisecond for the PICAXES at moderate clock speed). It has some instructions that take longer to execute than the period of time *t_INT* that is available to finish execution, turn to the interrupt routine, and wait for the message frame. Examples of such instructions are:

- SLEEP, NAP (Note that PAUSE and WAIT can be interrupted and thus may sometimes be preferred for use)
- READTEMP, READTEMP12
- SERIN, SEROUT, SERTXD
- SOUND
- COUNT

For example, a long instruction might end just after instance B in Figure 3, and when the network input line gets low during message bit transmission between instance B and C, it will erroneously turn to the interrupt routine, and will be out-of-synch with the transmitted message frame as well as with all future messages. Without precautions the node may even get blocked by a waiting SERIN instruction.

2) Uninterrupted program execution (only allowed for a finite period!): This may be needed if for example data is to be recorded continuously, or inputs have to be monitored without interruption.

The solution applied here is to introduce two subroutines that can be used to encapsulate the code part into a non-interruptible sub-program that is synchronized with the network upon exit:

- deSynch: This routine disables interrupts, allowing the target code to end without erroneously responding to an interrupt when a message transfer has happened simultaneously. As a result of the call to this routine the node gets de-synchronized with the network, i.e. processes on it do not catch any network messages anymore!
- reSynch: This routine monitors the network input line for a high condition, and if no message transfer occurs in a given time-interval interrupts are re-enabled. Consequently the process is properly synchronized with the network again.

A code part in a process may be encapsulated as follows:

```
REM PROGRAM
…
GOSUB deSynch
{ Code part or slow instruction }
GOSUB reSynch
…
```

Note that a consequence of this approach is that there is no guarantee that a process that is addressed through a message frame will actually get this information (and optionally respond). This is an import issue but in most cases does not cause severe problems. In many cases where a node acts as a sensor, the sensor will be queried for information on a regular basis, and a miss will be followed by a successful subsequent query. The best way to guarantee regular access to the node is to ensure that the encapsulated part of the process code takes relatively less time than the non-encapsulated, interruptible part. This can be done even by including an extra (interruptible) PAUSE command, as shown for example in the simple example in Section 10.1.

Another way is to add some "software handshaking" by requiring that a called process responds (and until that moment the process is queried repeatedly). Although perfectly feasible, this approach may consume considerable code space.

If the timing of the encapsulated code block in a process is approximately known, this information may be used to synchronize other processes with it as well. This approach will often be the most effective. For example, a process can be used to initiate a temperature measurement with a DS18B20 sensor. As it is known that a temperature measurement with this sensor takes a maximum of 0.75 seconds, another process that reads the measured value can be activated after this period.

Yet another way is to have a separate PICAXE-08M2 that focuses completely on network message handling, a second PICAXE at the same node taking care of the functional processes. A simple handshaking protocol based on polling can then be defined between these controllers. Consequently the network picaxe is always ready to respond to interrupts, as long as communication with the functional picaxe is performed only directly after a slave response message has been handled (i.e. those messages that do not create a timeslot, since a message will then never follow directly afterwards). Although this may look more complex, the low price of a PICAXE-08M2 may justify this effective solution.

6. Hardware implementation

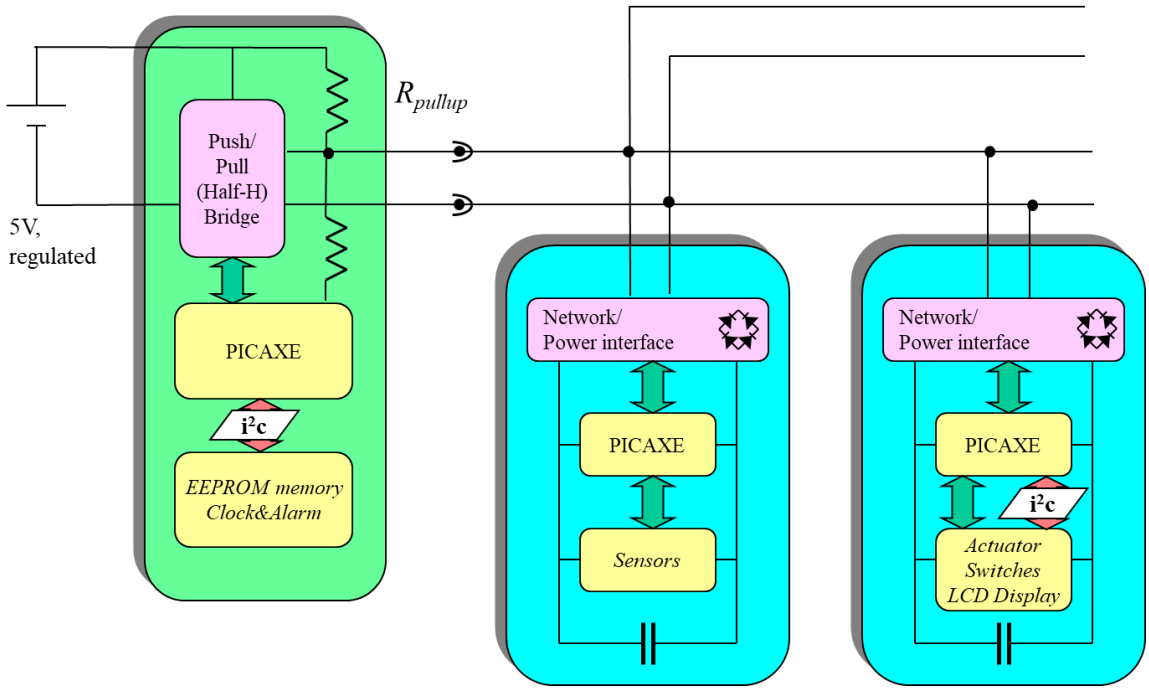


Figure 4: Hardware view of the network concepts

Figure 4 shows the main components of the network from a hardware viewpoint:

- A **master node**: responsible for power management and network configuration (through a push-pull active bridge) and the provision of short timeslots (during which the active bridge is replaced with a passive pullup) to allow for the slave nodes to send messages. The node may be extended with functionality that is useful for all processes, for example time provision, or memory capacity.
- One or more **slave nodes**; each slave node has an electrolytic capacitor as a primary back power source during communication timeslots. Furthermore a network interface is present that allows the slave to send as well as receive serial data frames. The slave's network interface uses a rectifier bridge based on Schottky diodes with low forward voltage drop in order to generate local GND and V_{CC} levels from the network. The local capacitor needs to be dimensioned according to the node power requirements as well as the duration of the passive pull-up network state. Note that a slave node may locally implement it own I^2C or SPI based serial network for communicating with devices (as shown in the left-most slave node)!
- The **network** wires; a simple two-line connection that is non-polarized; *slaves and master nodes can be connected in any fashion*. The network carries both power and communication information. Although unimportant from a practical

point of view, it is worth noting that it is the master connection to the network that defines the actual network polarity.

The network can be built according to any topology, as the network impedance (through R_{pullup}) is quite low and the communication speeds are low as well (2400 - 32600 baud).

All nodes receive messages sent on the network. Before any message is sent, the network is pulled low by the sending node, generating an interrupt on all other nodes and initiating a listening and decision process.

6.1 Master Node

The master node is responsible for power delivery to all slave nodes, the provision of timeslots to processes for communication, and may implement functional processes itself as well.

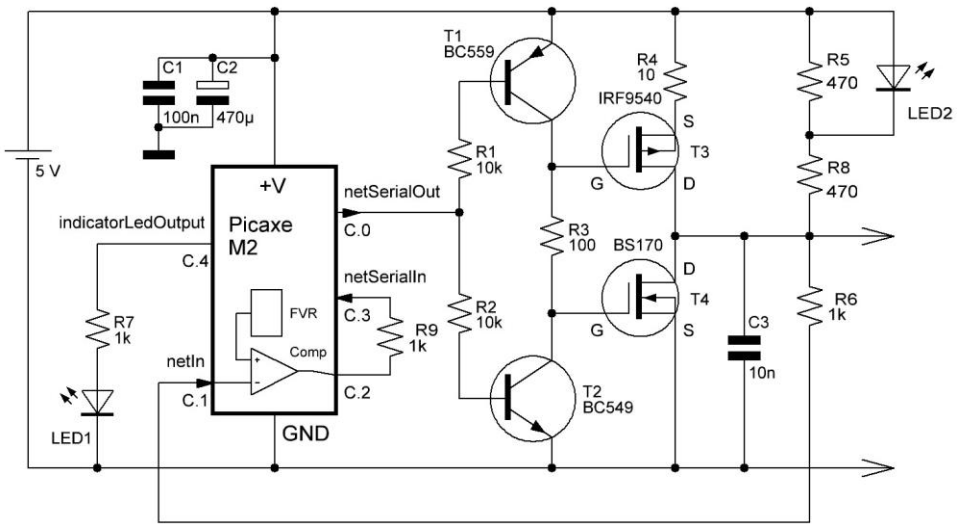


Figure 5: Basic master node circuit. See text for component values

The master node in Figure 5 consists of three basic building blocks:

- A PICAXE controller controlling the bus as well as (optionally) implementing functional processes.
- An active push-pull driver (or half-H bridge) consisting of a P-channel power MosFET (T3) and a N-channel small-signal MosFET (T4), allowing the voltage difference between the network lines to swing between Vcc and GND, as well as provide for a very low impedance power source to the network nodes.
- A transistor driver (T1, T2) to translate the CMOS voltage levels from the PICAXE to the levels required by the MosFETs (this is particularly important as

power MosFETs generally have a large gate capacitance and need a gate voltage of more than 4V in order to have a low drain/source on resistance (R_{DSon})).

The MosFET driver circuit shown here three-states the MosFET bridge if the PICAXE output is three-stated (which is the case if the corresponding pin gets defined as an input). Thus a separate pin for controlling the driver is not needed.

The master PICAXE communicates with the network *through* the MosFET bridge. This implies that during master-to-slave communications the network is powered when a HIGH (logic 1) is transmitted. Thus it is advantageous to transmit the value of 255 (all logic 1 bits) for data bytes that are not used.

R4 is inrush current limiting resistor that may be needed for large networks. R5 is the network pull-up resistor that provides for a high network level during communications. The PICAXE reads network information through the current limiting resistor R6.

The capacitors C1 and C2 are decoupling and power smoothing capacitors.

The following table shows values for the components that have been successfully applied during testing:

- IC1: PICAXE-08M or other microcontroller
- T1: General purpose small-signal PNP Transistor, BC559 etc.
- T2: General purpose small-signal NPN Transistor , BC549 etc.
- T3: P-gate Power MosFET, IRF9540N
- T4: small-signal MosFET, BS107, BS170 etc.
- C1: Decoupling capacitor, 0.1 uF
- C2: Power smoothing capacitor, 1000 uF
- C3: Signal Slope smoothing capacitor, 10 nF
- R1,R2: Base resistors for T1/T2, 10K ohm
- R3: Gate separation resistor, 100 ohm
- R4: Network inrush current limiting resistor, 10 ohm
- R5: Network pull-up resistor (1) , 470 ohm
- R6: Current limiting resistor for PICAXE serial input, 1K ohm
- R7: Current limiting resistor for LED, 470 ohm
- R8: Network pull-up resistor (2) , 470 ohm

According to figure 4, the master node may be locally extended with other components through an I^2C bus to implement processes like for example time provision, message storage or message display.

As a final statement it can be said that any PICAXE M2 or X2 type can be used. Since the primary task of the master node generally will be to provide for power and time slots, a PICAXE-M2 will generally be the optimal choice.

6.2 Slave Node

A basic slave node consists of three parts (Figure 6):

- A PICAXE controller implementing the serial interface as well as executing functional processes that may interact with sensors, actuators etc. through am I/O interface.
- A network interface consisting of a four-diode bridge rectifier generating local Vcc and GND levels, plus a fifth diode (D5) that provides a one-way isolation of the node's supply from the communication interface. All diodes are Schottky types that have a low forward voltage drop as well as a very low reverse current.
- A communication interface consisting of the bridge rectifier, R1, R2 and a small-signal MosFET T1.

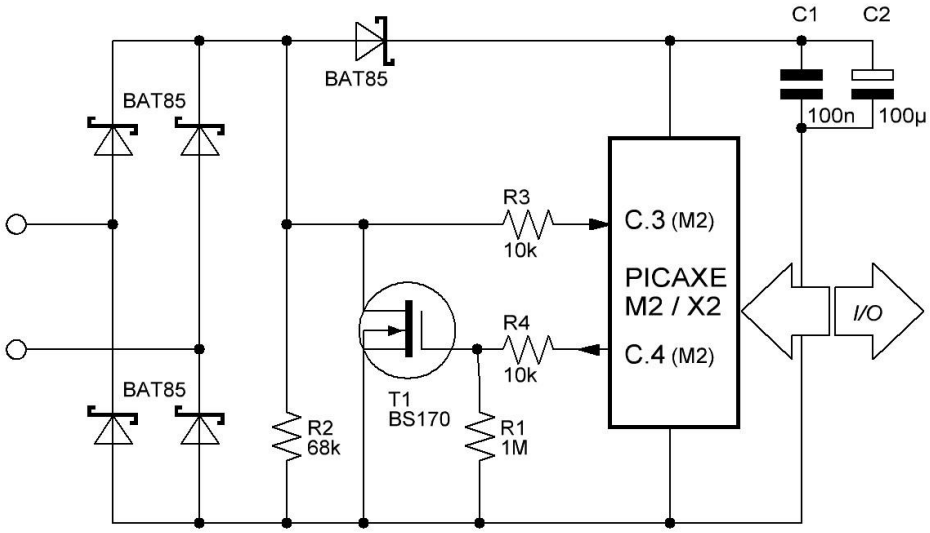


Figure 6: Basic slave node circuit. See text for component values

When the master node creates a voltage difference between the network connections through its push-pull bridge, the backup supply capacitor C2 is charged. Also a HIGH logic level is created at the PICAXE's network input. This logic level is maintained even when the active bridge is 3-stated and only the passive pull-up resistor of the master node is active. As soon as the voltage difference becomes small, because some node short-cuts the network lines, logic LOW level is read at the network input because R1 is pulling it low.

The slave node takes control over the network through switching via MosFET T1. Also a serial message frame is sent to the network through T1. Note that the signals sent by the

PICAXE are inverted by T1, and therefore the T2400 level is used for serial communications.

During testing the following component values applied successfully:

- IC1: PICAXE M2 / X2
- T1: small-signal MosFET, BS107, BS107A, BS170 or other with low gate threshold voltage (max $V_{GS(Th)}$ less than or equal to 3V)
- D1-D5: Schottky diodes with low forward voltage drop and low reverse leakage current, BAT85
- R1 Gate GND Level resistor, 1 Mohm
- R2: Input signal GND level resistor, 68 Kohm
- R3, R4 Current limiting resistors, 10K ohm
- C1: Decoupling capacitor, 0.1 uF
- C2: Power backup capacitor, 100 – 2200 uF

The required size of the power backup capacitor C2 depends on various factors:

- Power consumption of the slave node
- Size of the message frame (number of bytes)
- Network communication speed
- MosFET minimum gate threshold voltage
- Network pull-up resistor value

In the tested configuration having a slave node with only one led a 100 μF capacitor appeared already sufficient, it is wise to do some experimentation regarding its minimum allowable value.

Note that a voltage drop exists over the rectifier bridge (0.5 – 0.8V, depending on loading). This implies that the node local supply voltage will generally be somewhat less than 4.5V. Most of the modern ICs are specified to work as low as 2.7V – 3.0V, so this does not pose any problem. Some older ICs like the DS1307, IR receivers and most LCD displays need at least a 4.5V supply. Note also that the master node has full 5V supply.

The test configuration I used is depicted in Figures 7 and 8, implementing the example as described in section Section.1. After power-up the master node registers a process on the slave node that continuously reads a switch. If the switch has been pressed, a flag is set. Upon request by the master node -- and only if the flag has been previously set -- the slave node sends a message frame to another process on the master node to flash a LED. This simple application is sufficient for testing all bi-directional communication aspects. The test has been successfully applied using an in-house 20m network wire.

Figure 7: Prototypes for the master node (left) and a slave node (right), both based on a PICAXE-08M.

Figure 8: Test bench configuration with a short network wire. In the final successful test the network wire length was 20m.

The prototyping system used for testing is the famous Philips experimentation set of kits (Philips EE) as described on http://www.kranenborg.org/electronics . The circuit board layouts almost exactly follow the circuit diagram layouts.

7. Simplifications & extensions

7.1 Simplified slave node with read-only processes

If a node does not implement any process that sends to the network, the slave node network interface can be simplified by removing the MosFET transistor. This leaves us then with the reduced circuit as depicted in Figure 9.

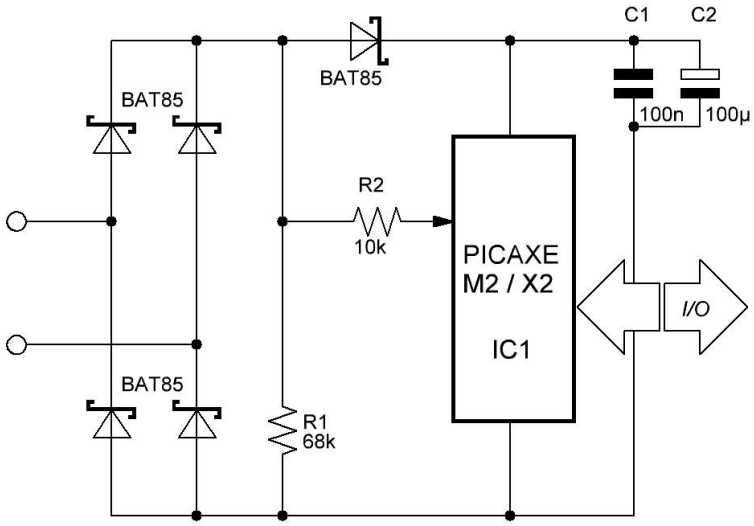


Figure 9: Slave node circuit that implements read-only communication. See text for component values.

7.2 Polarized slave node connection

The non-polarized network interface offers an extremely simple way of connecting, and this set-up is used generally in this document. In most applications however, a polarized interface is very acceptable (the polarity of an active network can be determined very easily). As a result we get a very simplified interface as presented in Figure 10. Apart from a reduction in components, the voltage drop over the interface is less (only 0.2 - 0.3V), leaving a larger local Vcc, which can be beneficial for components like LCD displays and other components with a minimum operating voltage of 4.5V. Additionally,

the logic levels generated at the network through switching via the small-signal MosFET are closer to the master node GND and Vcc levels.

The corresponding circuit diagram is given in Figure 10 below.

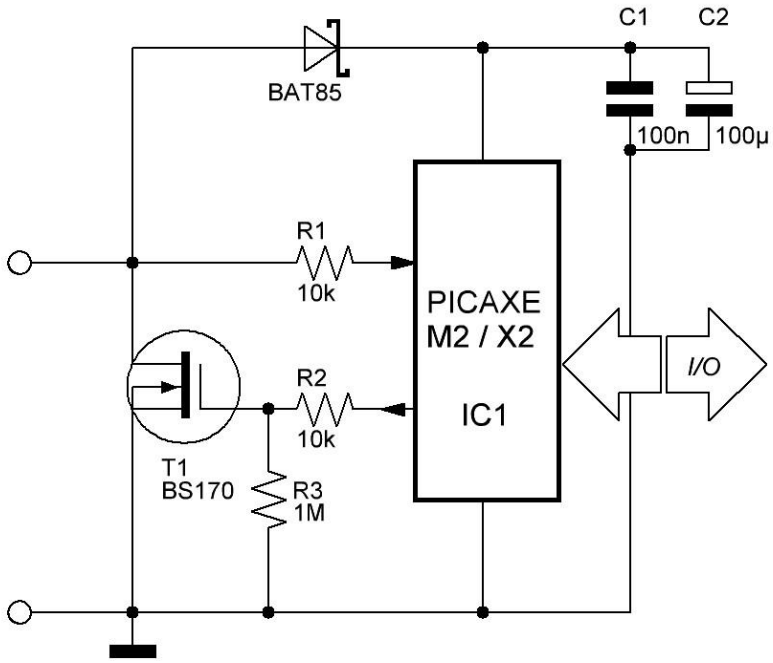


Figure 10: Polarized slave node connection

The following values apply successfully:

- D1: Schottky diode, BAT85
- T1: small-signal MosFET, BS107, BS170 etc.
- R1, R2:, Current limiting resistor, 10K ohm
- R3: Gate grounding resistor, 1M ohm
- C1: Decoupling capacitor, 0.1 uF
- C2: Power backup capacitor, 100 – 2200 uF

7.3 Simple network with separate power and communication lines

The “SerialPower” network concept can be applied to any physical network implementation, including implementations that have separate power and communication lines. The most simple type of network is the “diode-mixing” type (see [5] for a thorough discussion) in which a common pull-up resistor keeps the network at a logical high level and a logical low level is created by pulling the network low via the diode. In the latter case, the pull-up resistor limits the sink current through the diode and the microcontroller.

Figure 11 presents such a network that is valid for any type of microcontroller.

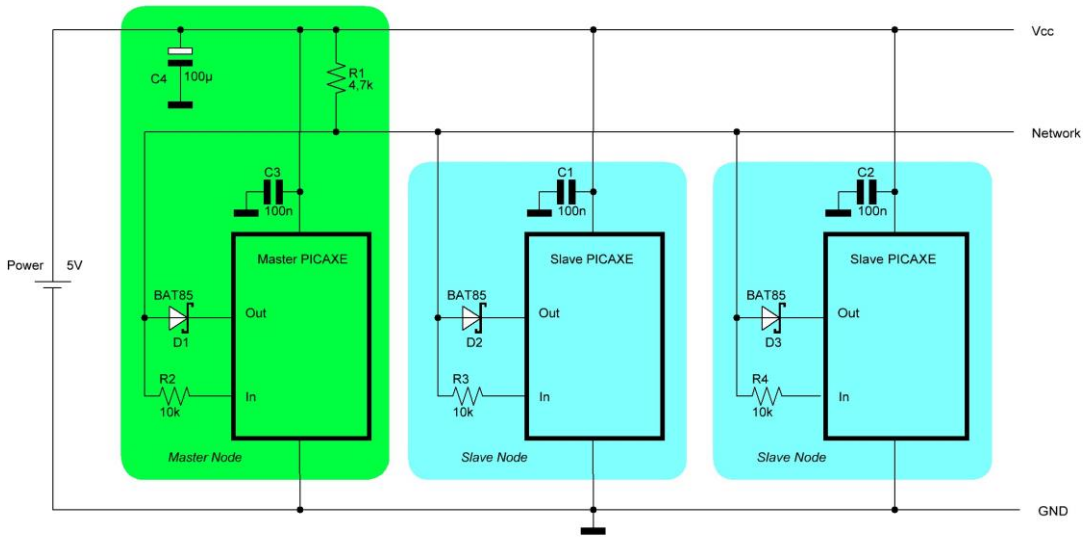


Figure 11: Simple diode-mixing network

Appropriate components for this configuration are:

- D1 - Dx: Schottky diode, BAT85 etc.
- R1: Network pull-up resistor: 1K – 10K ohm (determines network impedance)
- R2 – Rx: Current protection resistor in case of programming error, 10K ohm

The network software discussed in Chapter 9 can be applied here as well if some modifications to the slave network stack are made. These are needed since the sending slave now does not have a MosFET that inverts the output signal:

- Directly after power-up all network outputs should be set HIGH (master and slaves).
- The polarity of the interrupt signal should be reversed.
- The polarity of the SEROUT message in the `availableTimeSlot` routine should be reversed.

Implemented in this way, full bidirectional communications between processes on different nodes without message collisions is possible, even on the simplest physical microcontroller networks.

The appropriate adapted software implementations for the foregoing cases can be found at http://www.kranenborg.org/ee/picaxe/twowirenetwork.htm .

7.4 Plug & Play slave nodes (UNTESTED)

When a Plug & Play node is connected to the network, it should not disturb the operation of the network. Thus the primary function of a hardware solution is to control the impedance of the slave node as seen by the network. The circuit in Figure 13 is an extension of the slave node circuit in Figure 6 and can be described as to work in to stages:

Stage 1: When the slave node is connected to the network, the node impedance must be high enough to not disturb the network by pulling it to a low logic level, i.e. the connection should not lead to a voltage drop of the network below the minimum voltage level required for a high logic state. The worst case situation occurs when the network itself is at relatively high impedance, i.e. during a timeslot when only the master node pull-up resistor keeps the network at high logical level. During this stage the switching transistors T2 – T4 are not conducting (as well as T1) and the main backup capacitor C4 slowly charges through resistor R1 (whose value must be much larger than the master node pull-up resistor). During charging the impedance seen by the network increases because the capacitor charging current decreases. This allows at some point the slave node to be fully exposed to the network.

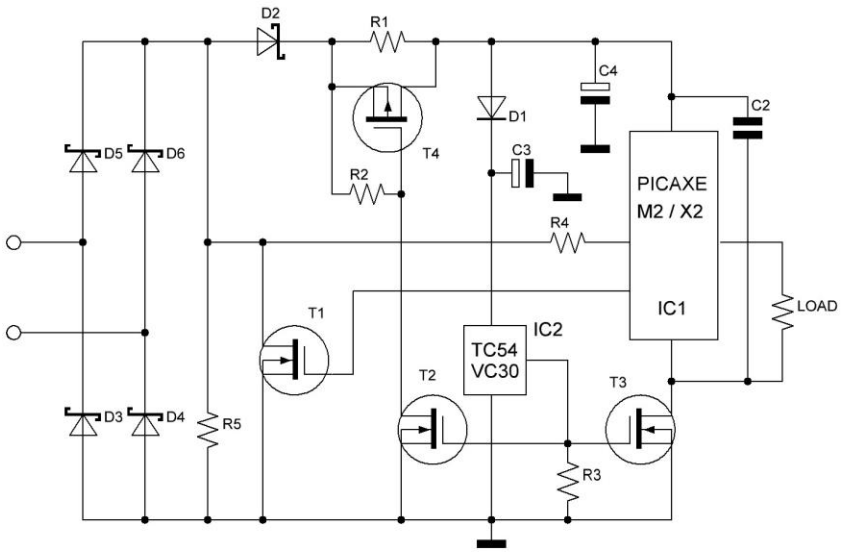


Figure 13: Plug & Play slave node hardware

Stage 2: When the backup capacitor has charged to a level that the potential difference over it is equal to about 3.7V the node can be exposed to the network by closing the T4 switch (through T2, causing R1 to be bypassed) and starting up the microcontroller (through T3). A Microchip TC54 voltage detector is used for quick closing of the switches T2 and T3 to guarantee a proper voltage rise time for the microcontroller, in particular for the 18X which does not have an internal brown-out function. The actual voltage level at the network interface will be a little bit over 4.0V due to the voltage drop over the rectifier diodes. As the backup capacitor is not fully charged at the beginning of this stage, the microcontroller should execute a sleep/nap instruction to reduce its power consumption to allow quick charging of the capacitor to its maximum level. The combination D1, C3 prevents resetting of the voltage detector due to start-up of the microcontroller and a possible associated voltage drop. At the end of this stage the backup capacitor is fully charged and the node is ready for operation.

In order to guarantee a proper low impedance network, we assume that the master node has a passive pull-up resistor with a value of 1 kOhm. Then, proper values for the components in Figure 13 are:

- D1: Ordinary silicon diode: BA318, 1N4001 etc.
- D2-D6: Schottky diodes with low voltage drop and reverse current: BAT85
- T1-T3: small-signal N-channel MosFET, BS107, BS107A, BS170 or other with low gate threshold voltage (max $V_{GS(Th)}$ less than or equal to 3V)
- T4: P-Channel power MosFET (IRF9540 etc.)
- R1: Slow-charging resistor, 4.7 KOhm
- R2: Gate resistor, 1MOhm
- R3: Gate resistor, 1MOhm
- R4: Current limiting resistor, 10 KOhm
- R5: Grounding resistor, 1MOhm
- R6: Curent limiting resistor, 10KOhm
- C2: decoupling capacitor, 0.1 uF
- C3: Back-up capacitor for voltage detector; 22 uF
- C4: slave node backup capacitor, 100 - 2200 uF
- IC1: Slave node PICAXE (any type)
- IC2: TC54VC30 (Voltage detector, 3.0V trip, active driver output)

8. Usage issues & performance

Catching events during message processing

When a message appears on the network, the nodes get involved in processing it during a time period at least equal to *t_comms*. This may mean that some very fast events during this time may be missed by the PICAXE controlling the node, and this effect counts stronger if the network traffic increases. A possible first solution to this is to use the SR-Latch functionality to capture these events and then subsequently check its state. Another option is to designate a separate PICAXE for continuous data processing, and to establish a simple handshaking protocol (based on polling) with the network PICAXE on the same node.

I2C versus SPI for I/O interfacing

Concerning the communication interface between the PICAXEs and peripheral ICs on a node, there exist the I^2C and SPI buses as the main synchronous bus options. The main characterization of these buses is given as follows:

- I^2C: flexible but slower than SPI:
  - The I^2C protocol includes device addressing, which implies that hardware selection of these devices is absent, and devices can be added without extra hardware.
  - Since the address detection takes some time, the protocol is somewhat slower than SPI, (generally 400KHz), but still fast enough for almost all practical applications.
- SPI: Faster dan I^2C, but less flexible if more than one device is used on the same bus:
  - The SPI protocol generally does not include device addressing (although there are a few important exceptions, see below), as a separate CS (Chip Select) pin should be available for each SPI device. Data is simply clocked in or out, which can be done very fast (several MHz).
  - Since a separate CS line is needed for every device, adding devices means also adding extra logic and consumption of extra PICAXE I/O pins.

Since all modern PICAXEs support the I^2C bus directly, the most efficient and elegant way is of course to use it. There are many interesting I^2C interfacing chips and devices that offer a multitude of I/O or processing capabilities, like the MCP23008/23017 I/O interface chips, the MAX6956 LED driver, 24LC512 EEPROM, DS1337 clock/calendar/alarm and many, many more devices.

In some cases it might be proficient to implement a SPI protocol instead of an I^2C protocol, as SPI is able to operate at higher speeds. Note that most peripheral chips are available in both I^2C and SPI variants.

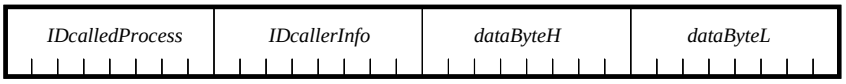
A few recent SPI variants of I/O drivers (MCP23S08/23S17) do include addressing in the communication protocol, a la I^2C. This implies that multiple ICs of this kind can share a single CS line. As a result, a configuration may be feasible that has several MCP23S17 devices (each device containing 16 I/O channels, all sharing a single CS line).

Note that most I/O drivers (I^2C and SPI) have a separate INT output (often open drain so that they may be tied together) as well as an interrupt register that may be polled by the PICAXE indicating that something on the input has changed.

9. Network Stack: Intelligent master node & auto-registering slave nodes

This chapter describes the network stacks for both the master node and the slave nodes. **You should first consult the User Guide (Chapter 11)** . The code is available for download from http://www.kranenborg.org/ee/picaxe/twowirenetwork.htm and is self-documenting. The available processes and their parameters are described in detail in this section. Note that in the codes a provision for 8MHz operation is included (but consult the User Guide, Chapter 11, on some cautionary remarks).

The following 4-byte message format is always assumed in this implementation for these processes:



9.1 Intelligent, network roaming Master Node network stack

The main goal of the master node is to provide for power and for timeslots on request for sending slave processes. In order to do so, the master node implements a main program that can be described using the following pseudo-code:

```
Initialize:
        Network Power-Up
        Configure comparator for signal processing, user-configurable parameters
        ROAM for sending processes (request process IDs from slave nodes that
                                    implement sending process, and store them
                                    in a RAM table)
loopCreateTimeSlots:
        DO (infinitely)
                Set pointer to start of RAM table with registered IDs
                DO WHILE not end of RAM table
                        GET next ID of sending process from RAM table
                        SEND NetworkMessage(availableTimeSlot, ID)
                LOOP
                IF plug&play option enabled
                        SEND NetworkMessage(roamForPlug&PlayNode)
                ENDIF
        LOOP
```

Thus, the master node continuously fetches a process ID of a registered process from the RAM table and subsequently creates a timeslot for it using an `availableTimeSlot` message, allowing the slave process to send a message to the network directly afterwards. Since the `availableTimeSlot` message contains the registered process ID as well, any message collisions can be completely avoided.

This code is executed in the body of the program and gets interrupted during a timeslot whenever a slave process sends a response to an `availableTimeSlot` message from the master. In the interrupt routine the slave message is decoded and may lead to the startup of one of the master node processes described further on in this section. One of these is `registerSendingprocess` causing a process ID to be added to the RAM table for timeslot generation for the corresponding process. In addition, user processes may be added as well.

Note that in an `availableTimeSlot` message the master node provides more than just the ID of the process that is allowed to use the current timeslot; it also provides "administrative" information (current maximum ID in the system, and location of the timeslotted ID in the RAM table) that in general is of little use to normal processes but which can be used by special nodes like the Network Manager Node and Plug&play nodes; see the next subsection on slave nodes for a definition.

The master node requires very little configuration in the software; in the User Area part of the symbol declarations, the following parameters can be adapted:

- Configuration data for Network Manager Node (can generally be left unchanged)
- Indicate whether plug & play nodes may be used (implying special timeslots, de-comment code line if this feature is to be used)
- Time between two timeslots (default = t_comms which should be regarded as a minimum, but may be set much larger for low-traffic networks). Note that this could even be made programmable by defining an extra process that adjusts this variable based on data in the corresponding message frame.

Since the network stack is defined such that the master roams for sending processes, the slaves respond automatically by registering theirs and the master node administers the registered processes, no user intervention is needed and the master node becomes essentially application independent.

In order to be able to add process IDs of processes that need timeslots to send messages on the network as well as to remove these IDs, the master node implements the following processes that can be addressed by any slave process:

- `registerSendingprocess`
- `unRegisterSendingprocess`
- `flashLEDmasterNode`

`registerSendingprocess:`

This process is called by a slave process whenever a message frame is put on the network during a timeslot where the following parameter definitions apply:

- IDcalledProcess = IDregisterSendingProcess
- IDcallerInfo = ID of the sub-process that requests a timeslot for a specific (different) ID
- dataByteH = Largest ID that a requesting slave node implements (including non-sending slave processes)
- dataByteL = ID of the process that needs timeslots (different from IDcallerInfo)

This process is invoked by slave nodes when the master node roams the network during network power-up and the slave nodes want to register sending processes. Furthermore timeslots can be requested by already registered processes on the fly (because of some sensor value being reached or switch being pressed requiring action), as well as by Plug & Play nodes after a response to a special `roamPlugNplayNode` message.

Note that the requesting process ID is different from the process ID that will get registered and receive timeslots; the requesting process at a slave node should already have timeslots assigned in order for this message to be send to the network!

The specification of the largest ID on the slave node that sends this message is used by the master node to determine the largest ID currently known in the network. This feature allows Plug & Play nodes to request for new, larger IDs that will not interfere with other processes already available in the network. Note that this maximum number must be based on all IDs, i.e. both sending and non-sending processes on a node. Of course a Plug & Play node is not required to used this feature; it may use and respond to existing network processes as well, but the feature allows new functionality to be added in a very flexible way using new processes that are unknown to the current network.

It is not possible to request timeslots for already registered processes; in that case nothing happens.

`unRegisterSendingprocess:`

This process can be used by either a slave process or the Network Manager Node to remove timeslots for a certain process; the corresponding ID entry in the RAM table is removed. The following parameter definitions apply:

- IDcalledProcess = IDunRegisterSendingProcess
- IDcallerInfo = ID of the process that requests timeslots for a specific ID to be removed
- dataByteH = Largest ID that a particular slave node implements
- dataByteL = ID of the process that gets un-registered and thus will not get timeslots anymore

Note that (in contrast to registration) it is possible for a process to un-register *itself*, since it has timeslots assigned to forward this message frame.

It is not possible to remove timeslots for processes that are not registered; in that case nothing happens.

`flashLEDmasterNode:`

This process can be addressed by any slave process to light the master node LED for a certain period of time, according to the following parameters:

- IDcalledProcess = IDflashLEDmasterNode
- IDcallerInfo = d.c. (don't care)
- dataByteH = d.c. (don't care)
- dataByteL = light period duration; in 2ms units

9.2 Auto-registering Slave Node network stack:

The slave network stack reads all messages that appear on the network and then subsequently calls the processes that are addressed in case they are actually implemented on the particular node. Central to this functionality is the interrupt handler which intercepts all messages, decodes them and then optionally starts up processes.

A slave node that implements at least one *sending user process* contains at least the following *system* processes:

- `roamSendingprocess`
- `availableTimeSlot`

`roamSendingprocess:`

This process is called by the master node after system power-up, to see which processes on slave nodes want to register for timeslots. Immediately after this command a slave node may respond with registering the process (because the master node generates a timeslot). The message frame from the master node contains a process ID that needs to be checked by the `roamSendingProcess` routine to see if it corresponds with a particular ID on the slave node that needs to be registered. The master node will send out `roamSendingProcess` messages for all allowable user process IDs separately, thus causing all sending processes to be appropriately registered in sequence. The slave network stack is configured such that after a `roamSendingProcess` message with a "hit", a registerSendingProcess message is automatically delivered for the master node, indicating that the particular process needs to be registered as a sending process and thus needs timeslots.

For this process (called by the master node) the following parameters apply:

- IDcalledProcess = IDroamSendingProcess
- IDcallerInfo = ID of the process that may get registered
- dataByteH = d.c. (don't care)
- dataByteL = d.c. (don't care)

The user needs to adapt two code parts:

1) Adapt the first code line of `roamSendingProcess` in order to let a number of processes to be registered. For example, if the slave node implements **userProcess1** and **userProcess2** that both want to have timeslots, the first line should be adapted as follows:

IF IDcallerInfo = **IDuserProcess1** OR IDcallerInfo = **IDuserProcess2** THEN

The subroutine will automatically send out appropriate `registerSendingProcess` messages with proper parameters for both processes.

2) In the user programmable area (SYMBOL declarations section), assign to `highestSlaveID` the highest ID of the processes implemented on this node (taking into account both sending as well as non-sending processes). With this information the master node can determine the largest user-specified ID currently available in the system, and thus grant new, larger IDs to Plug & Play nodes or other special processes in the system.

`availableTimeSlot:`

This process is called by the master node to indicate that a timeslot is available for a particular process, and all slave nodes network stacks should look:

- whether their node implements the particular process, and if so,
- whether the addressed process actually has a message to send.

Furthermore, if the addressed process has something to send the routine must assemble the corresponding message frame, take ownership of the network timeslot and subsequently forward the message to the network.

This process is called by the master node with the following parameters:

- IDcalledProcess = IDavailableTimeSlot
- IDcallerInfo = ID of the process for which the current timeslot is available
- dataByteH = Largest ID currently known in the system (both sending/non-sending processes)
- dataByteL = location of the time-slotted ID in the RAM table of the master node

If a certain slave process wants to forward a message with data to the network, it must write the bytes of the message to the following special RAM locations:

- RAMIDdestinationProcess ← Destination process addressed by slave process
- RAMIDsendingProcessInfo ← ID of sending slave process
- RAMdataByteLocationH ← byte data (MSB)
- RAMdataByteLocationL ← byte data (LSB)

With every `availableTimeSlot` message received the network stack inspects the contents of RAMIDsendingProcessInfo. As soon as a timeslot is provided for the ID in RAMIDsendingProcessInfo, the network software will construct a message frame from these RAM locations and put it on the network. After this, a value equal to IDnoprocess is written into RAMIDsendingProcessInfo, avoiding the message to be re-transmitted and indicating that the RAM locations can be used for new messages. Thus sending processes in the slave program body can inspect this RAM location in order to see when a new message can be constructed.

Note that the `deSynch` and `reSynch` routines need to be used in order to avoid interrupt during writing to the RAM locations. See chapter 10 for the code examples.

Adding user processes

User-defined processes can be added easily in the following way:

- Define a name and an ID for the process using SYMBOL definitions
- In the interrupt routine:
    - Add the ID to the list of IDs in the LOOKDOWN command (list between brackets)
    - Add the name of the process to the list of processes (between brackets) in the BRANCH command (using the sequence corresponding to the list in LOOKDOWN)
- Write the process routine:
    - As part of the interrupt routine, ending with the `GOTO exitFromSession` command
    - Mostly as part of the slave program body; the part in the interrupt routine merely passes parameters to memory locations and subsequently exits with the `GOTO exitFromSession` command.
- In case the new user process needs a timeslot, follow the instructions on `roamSendingProcess` at page 46.

Note that if a process only sends messages and does not have any parameter as input, it is NOT necessary to add its ID and address as a separate user routine or update the interrupt routine. See also the example in Chapter 10.1; the key reading process only resides in the main body.

9.3 Slave network stack for “listen-only” slave nodes

In case a “listen-only” slave node is used, the `roamSendingprocess` and `availableTimeSlot` routines can be deleted, as well as their addresses and IDs in the interrupt routine, leaving only user processes to be programmed.

10 Examples

Based on the "intelligent" master / auto-registering slave node network stacks described in Chapter 9, applications are developed in this chapter. All example code can be obtained from http://www.kranenborg.org/ee/picaxe . You may also consult the application development guide in Chapter 11 for the interpretation of the code examples.

10.1 Simple example: S1: push-button and led response

Here follows a very simple example application using an intelligent, network-roaming master node and one slave node, to demonstrate the basic features of the network concept. The slave node implements a process for reading a switch and subsequently flashing a LED, the master node has the `flashLEDmasterNode` process for lighting a LED when addressed. The example shows that bidirectional communication between nodes is possible.

When the network starts up, the master node first roams for sending processes by calling `roamSendingProcess` at the slave nodes. The process for reading the switch at the slave node (`processReadKey`) then gets automatically registered. Subsequently the master node regularly issues the `availableTimeSlot` command directed towards the switch-reading process, allowing `processReadKey` to put a message frame on the network each time the key has been pressed. This message frame contains the message "OK" and addresses the LED flashing process available at the master node (`flashLED`).

As a result of the above, the LED on the slave node flashes with each key press only if the key reading process has been enabled previously. Also the LED at the master node is lit only after the key-reading process at the slave instructs the associated master node process to do so.

With this simple application, all communication aspects of the network concept can be tested.

The code (Example S1) can be found on the SerialPower website on the software examples sub-page. Note that the code is extensively documented.

10.2: Remote temperature measurements (T1, T2)

The code (Examples T1 and T2) can be found on the SerialPower website on the software examples sub-page. Note that the code is extensively documented.

11. User guide for application development

The "SerialPower" network implementation for master and slave nodes as presented in Chapter 9 is very flexible, but the amount of available processes may look intimidating at first hand. Most of these though are system processes that are part of the software stack and are called automatically by the networking software. This short User Guide is to support development of standard applications without being exposed to all information.

First of all, the following issues are important regarding the type of network used (true two-wire or diode-mixed three-wire)

1. The master node software is identical for these network types. The only change that may be needed is a change of Picaxe type (in User-Programmable Area 1) and network node operation speed (in User-Programmable Area 2)
2. The slave node has a particulary important setting in User-Programmable Area 2 through the `REM #define simpleDiodeMixing` code line. The code is by default programmed for a true two-wire network. In case a diode-mixing three-wire network is used, the #define directed MUST be **un**commented (i.e. REM-statement removed)!

There are a number of general issues to be dealt with before starting programming the slave nodes:

1. What is the node configuration in the network; large master & tiny slaves, tiny master & large slaves etc? This issue is closely related to the information flow in the system; is it predominantly from master to slaves, or is slave-slave communication to be prevalent? See subsection "Node Configuration" on page 16.

2. Need some master setting nodes to be changed (generally not needed for tiny master nodes, i.e. those that only deal with timeslot provision and do not implement user functionality themselves) ? See Section 9.1, page 43

After these considerations one can start developing the applications through programming of the slave nodes. Slave node programming will be easy if you follow some guidelines:

Keep things simple:
In practice this means: Start with using not more than one sending process per node. The reason for this is that a sending process needs to fill four RAM memory locations (page 47) with databytes that are subsequently composed by the network stack to a network message frame. In case there are more than one sending processes, they have to compete with each other for these RAM locations and thus some form of synchronization is used. See the description of `availableTimeSlot` in Section 9.2.

Inspect the network stack and example software:

I put some real effort in code documentation, and the examples are meant to be instructive. Therefore it is suggested to download all code examples and have a look at them. The next step may then be to match the description of the network stack in Chapter 9 with the code examples. In order to be able to develop an application, you need to fully understand:

- How a message is received by a node and how a process consequently gets executed,
- How a slave process gets a message sent using the dedicated RAM locations,
- How processes can be registered automatically for getting timeslots.

Implement processes at the right place in the slave code

A slave node program always has the following structure:

SYMBOL declarations

Network Stack definitions (SYMBOL declarations)
User Area definitions:
- *SYMBOL & #DEFINE compiler directives*
- *I/O pins definitions (hardware interface)*

Slave initialization

Main Body:

DO
{Main Body Code}
LOOP

Network Stack:

interrupt:
{read network message}
{check if the addressed process is implemented on this node}
{jump to process if that is the case}

availableTimeSlot:
{check if a process on this node got a timeslot}
{check if this process has actually something to send (via RAM locations}
{if so, compose message from RAM locations and send message}

roamSendingProcess:
{check if the specified process ID(s) needs to be registered for timeslots}
{if so, send registration command for the indicated ID(s)}

deSynch, reSynch & checkReadyToPrepareNewMessage synchronization procedures

user routines:

user processes called as a consequence of the first message byte (IDcalledprocess)

END

The brown code parts generally need to be adapted for an application, the other parts should remain unchanged. The areas in which the code can/should be adapted by the user are clearly indicated in the code ("User Programmable Area X"! Note that the generation of a network message from the data in the RAM locations (availableTimeSlot) can happen without any user intervention.In general the following hints apply when adapting the previous structure to your application:

- A sending process (or several of them) executes in the main body (but may be addressed also via separate user routines, for example to start it up or to pass parameters to it)
- A non-sending (listen-only) process is best defined as a separate user routine (except when it takes very long to execute; in that case implement the user routine to simply pass parameters, and implement the main part in the body)

In many applications a sending process will run continuously in the background (and is therefore implemented in the main body of the slave) in order to monitor some sensors, switches or another device that indicates some condition. As soon as action is needed, this process fills the RAM locations (using the synchronization routines) with relevant data; the network stack takes care of the remaining actions to get the message put on the network. Note also that after a message has been sent, the network software indicates this (by writing `IDnoProcess` to the `RAMIDsendingProcessInfo` location), allowing synchronization of processes or new data to be sent.

Note that some simple process dispatcher is needed when there are several processes running in the main body.

Use the synchronization routines:
The synchronization routines are described in detail in Chapter 5 and play an important role in application programming. You should use them whenever:

- A "slow" instruction or code part is executed,
- A sending process composes the parts of a message to be sent by writing to the RAM locations,
- An instruction or code part needs to be executed at a different processor speed than the normal speed in the network. This case is encountered for example for "older" Picaxes when the nodes all run at 8 MHz but a certain instruction needs to be executed at 4MHz. Note that the newer X1/X2 parts perform clock switching automatically and thus generally do not need the synchronization routines.

Apart from these circumstances, it is often the case that multiple sensors need to be read at the same time. In that case, the synchronization routines can be used to prevent interruption between sensor readings.

Specify processes at a high abstraction level
Since a network message generally also contains the ID of the process that sent the message, the destination process can use the source ID to decide on how to use and

present information. For example, one could define a generic “Display” process running on a 18X node that can be used to display any type of information (even the node configuration in the system!).

12. Optimization options

[To be added in future sub-releases of this document]

13. Conclusions

The network concept presented here allows a simple but effective and practical form of distributed processing because:

- Power and data are distributed over just two wires that are interchangeable; only the master node needs a power source.
- Network communication is bi-directional, allowing communication between any pair of processes distributed over any number of nodes.
- The registration of processes that need timeslots happens fully automatic.
- All nodes are similar in that they catch all network messages and process them in a equal fashion.
- The network can be realized in a practical setting spanning several tens of meters.
- The protocol is efficient in that it allows small microcontrollers (PICAXE-08M) to implement both functional behavior as well as a full-fledged network stack.

The slave nodes in particular are very cheap as they use only a few standard components and the PICAXEs have an exceptionally good price/performance ratio. Consequently the network can also be very economically used with a master node that has another microcontroller than a PICAXE. This may open up a large potential of PICAXE applications in combination with existing, more expensive controllers.

Note that for a proper implementation of functionality a thorough understanding of distributed processing (in particular synchronization of processes) as well as a full understanding of the protocol and its implementation described in this document is essential.

14. References

[1] http://www.rev-ed.co.uk/picaxe/forum

15. Document and software update history

Version 1.0 (18 October 2006)
Original document, submitted for review on PICAXE forum
Version 1.1 (19 October 2006)
Polarized slave node interface (Section 6.2)
Version 1.2 (26 October 2006)
Alternative master node circuit for line driver without 3-state option, small errors corrected
Version 1.3 (April 2007)
Large revision: General introduction added, sectioning of document into parts 1 and 2 added, front page subtitle changed, new master driver circuit added (wilf_nv version), some alternative master circuits removed, change of input/output pins in example routines, small errors corrected.
Version 2.0 (September 2007)
Large revision: Intelligent network roaming master /auto-registering slave network stack added, plug & play nodes added, simple network hardware with separate power provision added, example applications added, User Guide for application development added.
Version 3.0.1 (April 2009)
Improved hardware diagrams, updated code for master and slave nodes in order to allow run-time selectable operation speeds (4 – 8 MHz)
Version 4.0 (October 2018)
Rewritten for implementation of modern PICAXE M2 and/or X2 variants (including crucial electrical circuit adaptations)