Control of a Brushless Motor Using a STM32 Electronic Board

ANA Avatar XPRIZE, A.A.: 2020/2021

Carlo Bosio

Sant'Anna School of Advanced Studies

April 6, 2021

1 Introduction

This project is part of the work carried out for the design of the Avatar platform for the ANA Avatar XPRIZE Competition. The Sant'Anna Team Avatar, for vision related tasks, is equipped with cameras, that are fixed on an anthropomorphic head. It is required to decouple the head and the Avatar's platform orientations in order to avoid motion constraints on the vision system. This is obtained by mounting two JMC brushless motors, that allow to control the head pitch and yaw angles. This project objective, then, was to develop the motors control structure and set up the interfacing between the motor driver and the PC. For this purpose a STM32 Nucleo F746ZG board has been used. The activity is described in detail in this report, that could serve also as "User Manual" to set up and configure the system (PC, Microcontroller and Motor). The activity consisted of two main phases:

- The Microcontroller set up and programming (with the dedicated software *STM32CubeMX* and the System Workbench based on *Eclipse*) supported with the communication scheme implemented on *Simulink*;
- The development of a *ROS* node for the communication between the PC and the Microcontroller (substituting the *Simulink* scheme).

1.1 The Motor

The motor used is a JMC 3 phase brushless motor accompanied with a digital AC servo driver. The motor is provided with an encoder characterized by 4096 different position per turn (and thus it will be possible to achieve an angular positioning precision of the motor axis of $360°/4096$). The motor-driver system needs a 24 V power supply.

The motor control strategy can be implemented in two ways. The first way consists of a high level control, based on the JMC dedicated software and the ethernet connection between the driver and the PC. The second way, that is the one adopted, is based on low level electrical inputs: a pulse train generated by a PWM and a simple logical signal. With this logic, a pulse rising edge corresponds to a rotation of the motor axis of 360°/4096 ($\simeq$ 0.0879°), whereas the simple logical value sets the rotation direction. Thus, it is possible to modify the motor axis angular velocity by varying the PWM period and to invert the rotation by switching the logical input.

1.2 The Microcontroller

The Microcontroller (MCU) used is a STM32F746ZG, integrated on a Nucleo-144 board[1]. The board requires 5 V USB power supply, while the communication with the PC is set up via the ethernet port. The board is provided with a dedicated software (*STM32CubeMX*) to configure the pins and generate, build and run the code for the MCU.

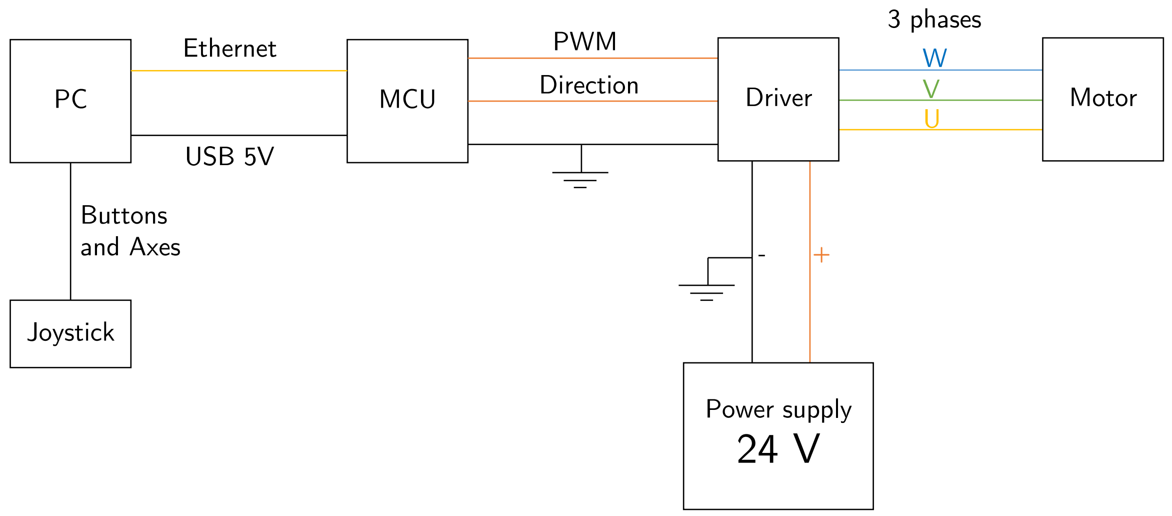

Figure 1: General scheme of the system

1.3 PC and Input Commands

In order to implement the communication between the PC and the MCU, a *Simulink* scheme implementing a UDP protocol has been created (later replaced by a *ROS* node for the implementation on Linux). For the input commands a standard USB Joystick has been attached to the PC. The general scheme of the whole system is represented in figure 1.

[1]Nucleo-144 User Manual

2 Set Up

2.1 MCU Set Up

As described above, for the motor control a PWM and a simple logical pin are essential. These two functions are obtained configuring a general purpose I/O pin and a Timer. Moreover, since for a position control it is required to count the pulse train rising edges, three Timers have been activated: two identical PWMs (one linked to the driver input and the other for internal counting purposes) and a "External Clock Mode" Timer (actively counting the incoming pulses incoming from the second of the two identical PWMs). For the Timer configuration several parameters need to be specified:

- The Prescaler (PSC): a 16 bit value scaling the internal MCU clock frequency (100 MHz in this case) to the desired Timer couning frequency;
- The Counter Mode: specify whether the Timer should count increasing or decreasing the count variable;
- The Counter Period (CP): a 16 bit value specifying how many internal prescaled clock pulses should trigger the count variable;

Using this notation, the Timer counter frequency is:

$$\nu_{\text{TIM}} = \frac{\nu_{\text{IN}}}{CP(1 + PSC)} \tag{1}$$

Where ν_{IN} represents the internal clock frequency for ta PWM and the incoming pulse train frequency for the "External Clock Mode" Timer.

Name	Pin	Type	From	To	Mode	Properties
Dir_1	PB8	GP I/O	–	DIR +	–	–
TIM3	PA6	Timer	–	PUL +	PWM	$PSC = 1$, $CP = 50000$
TIM4	PB6	Timer	–	TIM9	PWM	$PSC = 1$, $CP = 50000$
TIM9	PE5	Timer	TIM4	–	External Clock	$PSC = 0$, $CP = 65535$

Table 1: Pins configuration

The pins configuration is described in the table 1 while the electrical links with the driver input ports are represented in the scheme of figure 2.

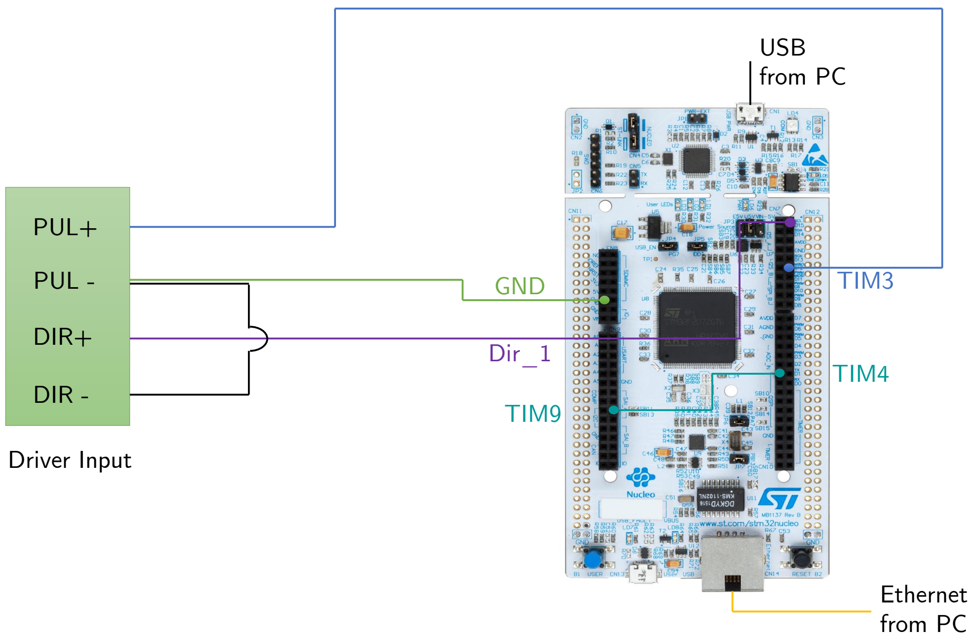

Figure 2: Electrical linkages scheme

2.2 Communication Set Up

In order to control the motor it is necessary to exchange data with the MCU. This is done by implementing a UDP communication scheme using a *Simulink* real time diagram (showed in figure 3).

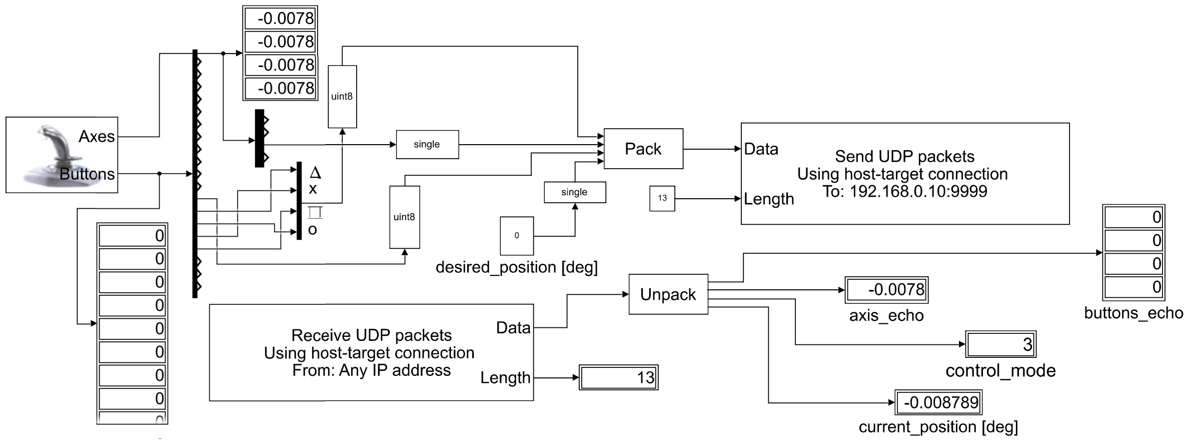

Figure 3: *Simulink* communication scheme

The establishment of the communication requires the configuration of the IP addresses and ports. In particular:

- The MCU IP address is 192.168.0.10 and its receiving port is the 9999;

- The PC IP address is 192.168.0.100, the sending port is the 25001 and the receving port is the 11111;

Clearly the PC address and ports depend on which PC is connecting to the MCU and the above numbers (that are the ones used in the code) refer only to the PC on which the work has been carried out.

In the sending part (from PC to MCU) of the communication scheme, Joystick data from five buttons and one axis are collected and concatenated to a user specified single precision floating point constant value (desired_position). The packet payload is represented in table 2

Bytes	1	1	1	1	4	1	4
Type	uint8	uint8	uint8	uint8	single	uint8	single
Object	△	×	□	○	right axis ↔	R1	desired_position

Table 2: Sending packet composition

In the receiving part (from MCU to PC), the packet has the same structure (table 3) and collects two motor state variables (control_mode or current_position) and the echos of the buttons and axis values (mainly for functionality check).

Bytes	1	1	1	1	4	1	4
Type	uint8	uint8	uint8	uint8	single	uint8	single
Object	△	×	□	○	right axis ↔	control_mode	current_position

Table 3: Receiving packet composition

The MCU code for data processing is explained in the next section.

3 MCU Programming

The MCU main functions consist of handling the data from/to the PC and define the motor operation modes. All these tasks are achieved in the attached code, whose principal functions are described below.

3.1 Receive Function

Incoming data are handled by `recv_fun`. This function exploit a local `char[13]` array (`inc_data`) to which the received data buffer payload (`datarecv->payload`) is copied. Then the received values are copied from `inc_data` to the two global variables `prova_int` (`uint8_t[5]` array) and `prova_single` (`float[2]` array). In this way the received data are stored in the local MCU memory and can be used for the following tasks. At the end of the execution, the input data buffer `datarecv` is freed and the global control varible `my_watchdog` is set to its maximum value. The watchdog variable is decreased by 1 every time the main task is executed. This increasing/decreasing mechanism of the watchdog variable makes sure that the MCU does not execute the main task if no input packet is coming.

3.2 Main Task

In the main task an infinite temporized loop is executed every 10 ms. Inside the loop there is a preliminar watchdog check, so that the following instructions are executed only if the MCU is receiving data (i.e. if `my_watchdog > 0`).

The operating mode is determined by the global `uint8_t` variable `control_mode`, that is increased every time the R1 button is pressed (its default starting value is 0). There are four possible operating modes, depending on the `control_mode` value:

- When `control_mode == 1` the motor is in "stepper" mode;
- When `control_mode == 2` it is in speed control mode;
- When `control_mode == 3` it is in position control mode;
- In all other cases it is in stop mode.

The variable check and the instructions execution is implemented in a `switch` statement. All the operating conditions are obtained by adequately varying the timer parameters (in particular period and duty cycle). In stop mode, for example, duty cycle of TIM3 and TIM4 is set to 0, in order that no pulse is generated from the PWMs. This is done by accessing the variable `CCR1` of the timer structure (e.g. `htim3.Instance->CCR1 = 0`).

3.2.1 Stepper Mode (`control_mode == 1`)

In Stepper Mode, the motor is desired to rotate "step by step", i.e. varying its angular position by discrete values. To obtain this behavior, the MCU has to send to the driver a user specified finite number of pulses. If n pulses are sent, the motor axis rotates by $\frac{n}{4096}$ turn. The number 4096 corresponds to the quantity of input pulses needed by the driver to make the motor axis turn by 360°. Since this is a nominal and recurrent number, in the code it is memorized with the `#define` variable `NPULSES`. In addition, it is realistic, instead of using the motor directly, to add a gearbox between the motor and the load. This is considered in the code by adding another `#define` variable called `GEARBOX_RATIO` (in the specific case the reduction ratio is 20). Consequently, in stepper mode, for each step of n pulses, the output axis rotates by $\frac{n}{NPULSES \cdot GEARBOX\_RATIO}$ turn.

To trigger a step it is sufficient to press the □ or ○ button, depending on whether the motor is desired to rotate in clockwise or anticlockwise direction. These buttons values are stored in the MCU memory in `prova_int[2]` and `prova_int[3]` variables) and are used to activate an auxiliary function called `motor_step`.

The `motor_step` function takes three arguments as input: `npulse` (`uint8_t`), `cp` (`uint16_t`) and `direct` (`uint8_t`). The current TIM9 counter value (`htim9.Instance->CNT`) is stored in a local variable `cinit`. It is then started an infinite loop inside which `cp` is assigned as TIM3 and TIM4 counter period value (and `cp/2` as duty cycle) and `direct%2` is assigned as Dir_1 pin value (the `%2` operator is just to avoid problems due to errors in the input assignment). To exit the infinite loop, the requested number of pulses has to be reached, so there is a control on the TIM9 counter value, that has to be greater than `cinit + npulse`. Actually this is not enough to guarantee the correct exit from the infinite loop, since the TIM9 counter could overflow. For this reason the overflow condition is spotted by checking if `htim9.Instance->ARR + npulse` is greater than the TIM9 counter period. Once the infinite loop is terminated, the step has been performed and the TIM3 and TIM4 duty cycle are set to 0. The function returns an `int` output (the only values allowed are ± 1 and depend on the step rotating direction `direct`) that will be used at the and of the main task infinite loop to compute the current motor axis angular position (and send it back to the PC).

In the attached code, pressing the buttons triggers a step of 100 pulses, with a period of 1000. This means that the step is performed in around $2 \cdot 10^{-3}$ s.

3.2.2 Speed Control Mode (`control_mode == 2`)

In Speed Control Mode, the motor is desired to rotate with an angular velocity directly proportional to the transmitted axis value (i.e. the right axis $\leftrightarrow$ motion, stored in `prova_single[0]` variable).

An initial check on the axis value is performed (if its module is lower than 0.1, no action is done). Depending on the axis sign, the direction is imposed writing the corresponding value on the Dir_1 pin. The TIM3 and TIM4 counter period is set making sure to not overcome the 3000 rpm angular velocity nominal limit (lower bound of 245) and imposing an upper bound of 50000. These two bounds are stored in the `#define` variables `P_MIN` and `P_MAX`. The counter period can be computed as:

$$CP = \frac{P_{MAX} \cdot P_{MIN}}{P_{MIN} + (P_{MAX} - P_{MIN})\frac{|x|-0.1}{0.9}} \tag{2}$$

Where x is the axis value. The function is represented in figure 4.

In this case again a ± 1 value is assigned in order to later compute the current output axis angular position.

3.2.3 Position Control Mode (`control_mode == 3`)

In Position Control Mode, the motor is desired to reach a user specified position and keep that position till another position is set. Therefore, for the sake of simplicity and effectiveness, a P controller is implemented in this block. In order to handle all the

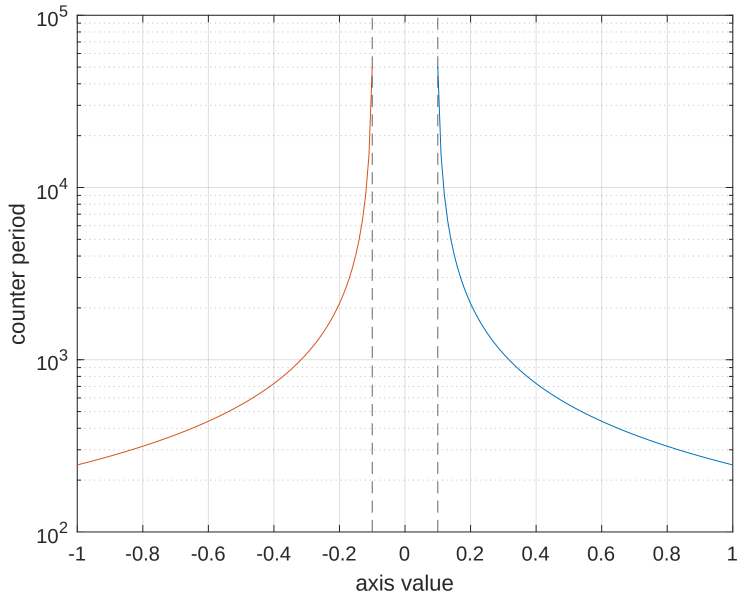

Figure 4: CP as function of x in semilog scale

position-related variables, a global `struct` called `position_cnt` is defined. Every time the MCU is started, a `position_cnt` entity called `p` is introduced and associated to the motor. This structure contains all the useful variables to handle counters, position (in degree and in pulses), direction and these variables overflow conditions. In particular, `position_cnt` contains:

- `uint16_t old_cnt`, `uint16_t new_cnt`, `uint16_t delta_cnt`: variables used to store the TIM9 counter values (relative to the previous `main_task` loop execution and the actual one respectively) and compute the position variation handling the counter overflow;
- `long int position_old`, `long int position_new`: variables used to store the position values (in pulses) relative to the previous `main_task` loop execution and the actual one respectively;
- `float deg_position`: variable storing the actual position value in degrees;
- `int direction`: variable that can assume only ±1 values. It describes the actual output axis rotation direction and is used to compute the actual position. It is in one-to-one correspondence with the Dir_1 pin value ($1 \leftrightarrow 1$, $-1 \leftrightarrow 0$);
- `float deg_desired_position`: variable that collects the user specified angular position (in degrees) to which the output axis should be brought.

At the beginning of the block, `prova_single[1]` value is assigned to `p.deg_desired_position`. Depending on the sign of `p.deg_desired_position - p.deg_position` the Dir_1 pin value is wrote. The P control constant is set in order to achieve a 90° rotation in 1 s, giving `K = 3` (stored as `#define` variable).

Exploiting an auxiliary function called `omegatoCP` (which takes as input the desired angular velocity value `omega` and the PWM timer `prescaler` and returns the corresponding counter period) a counter period corresponding to an angular velocity proportional to the position gap is set to TIM3 and TIM4. Also in this case 3000 rpm should not be overcome, so there is a saturation check that sets `P_MIN` as counter period if the computed value is lower.

At the end of the infinite loop block, the position is computed. The TIM9 counter value is stored in `p.new_cnt`, that is compared to `p.old_cnt` in order to spot the counter overflow condition and compute the position variation `delta_cnt`. At this point the `p.direction` variable (which is assigned in every block) comes in, and the current position is computed as:

```
p.position_new = p.position_old + p.direction * p.delta_cnt
p.deg_position = (float)(p.position_new)*360.0/(NPULSES * GEARBOX_RATIO)
```
(3)

Finally, to the `old` marked variables are assigned the current values and the next cycle can begin.

3.3 Send Function

Outgoing data are handled by `UDPCommunication`. At the beginning of this function, all the addresses and port required by the UDP protocol are defined. In the specific case, the MCU will send the packet to the 192.168.0.100 address, port 11111. This function, as well as the `main_task`, is based on a 10 ms temporized infinite loop. A 13 bytes payload buffer (`datasend`) is initialized and a local `char[13]` array called `dati` is exploited to temporary store the data. The outgoing data consist of buttons and axis echos together with the `control_mode` variable and the current `deg_position` value. The array `dati` is copied to the buffer payload and then `datasend` is sent and freed. Finally, `UDPCommunication` can send the next packet.

3.4 Second Task

In addition to the above described functions, a low priority task is defined. The only purpose of this task is to check the PC-MCU communication correct operation without exerting any action on the motor. In this task a 25 ms temporized infinite loop is defined and the main function is to turn on and off the three LEDs integrated on the board. A variable `light_status` is defined and is increased or decreased every time the buttons $\times$ and $\triangle$ are pressed (respectively). All the possible light combination are defined in a `switch` statement inside the auxiliary function `personal_switch`, that takes `light_status` as input and depending on its value turn on a combination of the three LEDs.

4 ROS Implementation

In this section the *ROS* implementation of the PC-MCU communication (from the PC side) is described. This implementation seems to be necessary because most of the time the code and the data exchange for robotic applications is not handled with *Simulink*, but with *ROS*. For this purpose, *ROS Melodic* in a Linux environment has been used. A *ROS* package called *smc_driver* has been created, in whose *src* folder two *Python* files are located:

- *SMC_driver.py*: where a class is defined in order to manage the operations performed on the driver;
- *SMC_UDP_comm.py*: the file effectively implementing the communication scheme.

Also in this implementation, a Joystick is used as input for generating the data to be sent to the MCU. For this reason, the *joy_node* is activated and the node (*Data_exchange*) defined in *SMC_UDP_comm.py* subscribes to the *joy* topic, calling a callback function every time a Joy message is received. The two nodes are launched by the *demo.launch* file.

4.1 Driver Class

This class does not represent a strictly necessary tool to manage the driver, but it is useful to organize the code in a rational manner.

When a `SMC_driver` object is defined, the constructor is called. It firstly sets the sender (PC) and receiver (MCU) IP addresses and ports. The default values are the ones involved in the specific case (the same IPs and ports used above) and need to be modified depending by the user. The other class variables correspond to the data that need to be exchanged (buttons, axis, position...). A differentiation is made between the parameters that has to be sent to the MCU (whose name begins by `send_`) and the variables that have to store the incoming data (whose name begins by `receive_`). The last variable is a socket, initialized with personl IP address and receiving port number, that will be used for data exchange.

Several functions are defined in the class. It is possible to print the object IP and port by the `__str__` and `__repr__` functions, or entirely print the object variable set with the function `print_attributes`. It is possible to set all the `send_` variables values and get the `receive_` variables ones. The `send_to_driver` function sends the `send_` values to the MCU via the socked (as well as was done by the *Simulink* “Send UDP packets” block), while the `receive_from_driver` function stores the current driver parameters in the `receive_` object variables, via the same socket.

4.2 Data Exchange Node

The `Data_exchange` node is defined in the `SMC_UDP_comm.py` file. At the beginning of this file, a `SMC_driver` object called `driver1` is initialized and an auxiliary global array `buttons_to_send` is defined. This array will be useful to assign the incoming buttons values from the Joystick. A `callback` function is defined to perform this assignment every time a "Joy" type message is received (by subscribing to the `joy` topic). In addition, the node subscribes to the `des_position` topic. This will be useful to input the desired position from the terminal by publishing a `std_msgs/Float32` message when the MCU is in Position Control Mode. The command has the following syntax: `rostopic pub des_position std_msgs/Float32 xxx` (where `xxx` is the desired postion value). When a message is received from this topic, the `callback1` function is executed, inside which the desired position value is set as `driver1.send_position` vaue. In the `Data_exchange` node main loop (1 ms temporized by the `rospy.rate(100)` instruction) two simple sending and receiving operations are performed. All the buttons functionalities are the same as described in the previous section. Finally, a global counter is defined in order to print on terminal the `driver1.receive_` variables once a second.

The `demo.launch` launcher allows to activate both the *Joy_node* and the *Data_exchange* node with a simple prompt instruction.