A. Yudin, M. Semyonov "Distributed Control System for a Mobile Robot: Tasks and Software Architecture" in Research and Education in Robotics - EUROBOT 2011, ser. Communications in Computer and Information Science, vol. 161. International Conference, Prague, Czech Republic, June 15-17, 2011, Proceedings: Springer Berlin Heidelberg, 2011. pp. 321-334.

Distributed Control System for a Mobile Robot: Tasks and Software Architecture

Anton Yudin and Mikhail Semyonov

Bauman Moscow State Technical University, IU4 Department,
2-nd Baumanskaya st., 5, 105005, Moscow, Russia
skycluster@gmail.com,m7onov@gmail.com
http://www.bearobot.org

Abstract. This article discusses several aspects of the distributed control system being developed for a mobile robot. The aim for the system is ease of use for educational purposes and flexibility for easier and faster robot development for new projects. Movement tasks are formulated for better understanding of the distributed program architecture. The network protocols necessary for the system's operation are discussed. The higher level protocol's concept being developed by the authors is provided and an example of the protocol use is presented.

Keywords: Eurobot, mobile robot, distributed control system, modular systems, network protocol, education.

1 Introduction

The practice of developing mobile robots for the Eurobot competitions [2] includes all the main stages of the production cycle of any modern high-tech device. In addition to project management participants of the team's development process are faced with research, design and development of mechanics, electronics, control programs, and finally with integration of individual system components into a single robotic solution. Bearing in mind that most of the competitors are students, the question of learning and acquiring new knowledge and skills is one of the most important.

Year after year, the competition offers developer teams a new task, which in turn leads to a complete revision of the mechanical structure of the previous robot. Mechanics when being changed influences electronics and control programs. A few years of robot developing is enough to notice that each of these areas may provide basic building blocks which will facilitate development of future systems. And if universality for the mechanics can be achieved only at the level of individual system components (e.g. motors, wheels, and fasteners), the electronics and the software can both be involved in a core of unchanged structure.

In this article, the authors attempted to summarize the experience of creating such a core system, while continuing to carry out one of the goals of the beArobot team they belong to: find and develop basic solutions for teaching and for use

in the competition. In the previous article [1] the purpose and the relevance of a distributed control system were justified, general structure of the system was represented, the principle of configuring and using the system was given, and the hardware implementation of the system was described.

2 The problem statement

To start with, let's present the hardware devices which will be used later on when describing the core of the distributed control system. In this case, we consider one of subsystems of a mobile robot, namely - a subsystem of relative navigation (Fig. 1)

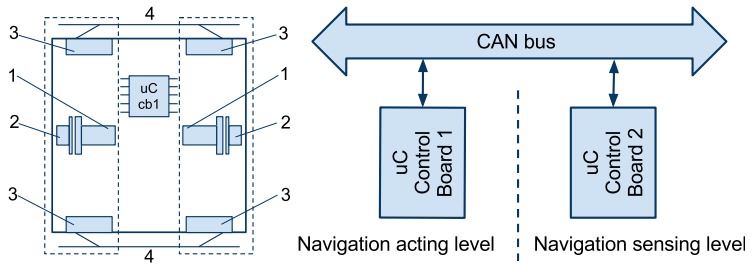

Fig. 1. The composition of the relative navigation subsystem. To the left: the structure of mechanical devices (1 - engine, 2 - sensing encoder, 3 - button, 4 - bumper). To the right: the configuration of the electronics on various control levels.

The figure shows relative navigation subsystem's two parts, which represent it physically. The mechanical structure in this case is sketchy on the location of the mechanisms, but nevertheless all the subsystem's necessary devices are present. The configuration of control electronics is reduced to two printed boards with microcontrollers that use the 8-bit AVR architecture. One of the boards is shown in the mechanical sketch of Fig. 1. Its functions include direct motion control of a robot's chassis for point to point movement tasks (in the figure: uC Control Board 1, uC cb1; further on cb1). The second board (in the figure: uC Control Board 2; further on cb2) coordinates a robot's navigation system: processes data from additional sensors, identifies obstacles, builds movement routes and sends commands to the first board.

The complete control system can be represented with 4 to 7 control boards, but in this case, for simplicity of the proposed approach's description, we confine ourselves to the two outlined above.

Now let's consider the actions that are available to the cb1 control board (Fig. 2). By its example, we hope to show the idea of how the proposed control system will be organized and how it will operate.

As can be seen in the figure, the cb1 board, in addition to liaison functions with the cb2 board, provides such functions as: indication of a bumper's state,

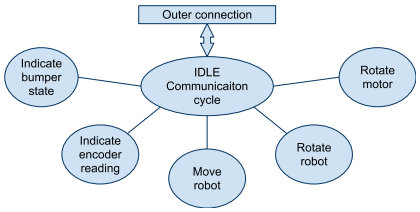

Fig. 2. The available actions of the cb1 control board.

indication of encoder sensors' meter readings, activation of the robot's linear motion, activation of the robot's rotation, activation of a single motor's rotation.

Next, we will consider the three figures (Fig. 4, Fig. 5, and Fig. 6) which demonstrate processes that occur on the cb1 board, as well as their interactions. For this purpose we introduce the following notation (Fig. 3).

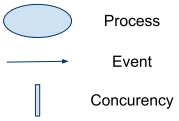

Fig. 3. The description of the graphical notation.

Fig. 4 shows[1] processes that the cb2 board can initiate on the cb1 board, and that are assigned to work with sensors onboard of the robot's chassis.

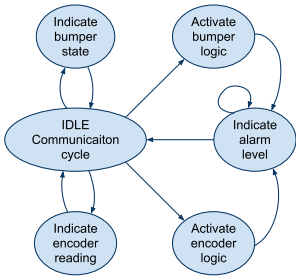

Fig. 4. The processes machine for the sensors on the chassis.

When activated, the processes can monitor the sensor's status automatically according to some local control logic (algorithm). In case a necessary change is recognized the board can reveal it by increasing the level of the network's alarm

[1] Fig. 5 and Fig. 6 also show the processes which can be initiated by the cb2 board.

in a certain way. Also on request, the processes can indicate the current status of sensors on the chassis.

Fig. 5 shows the processes that allow individual rotation of each motor's shaft. There are two possible control algorithms: the first rotates a motor's shaft at a certain angle, the other rotates a motor's shaft at a certain speed. According to the algorithms' start commands the board's microcontroller automatically begins monitoring bc1's sensor readings and using them in the algorithm. Also an algorithm that processes the state of a robot's bumper works in parallel with the basic algorithm and in case of a collision the level of the network's alarm is raised, resulting in immediate and automatic motor stop.

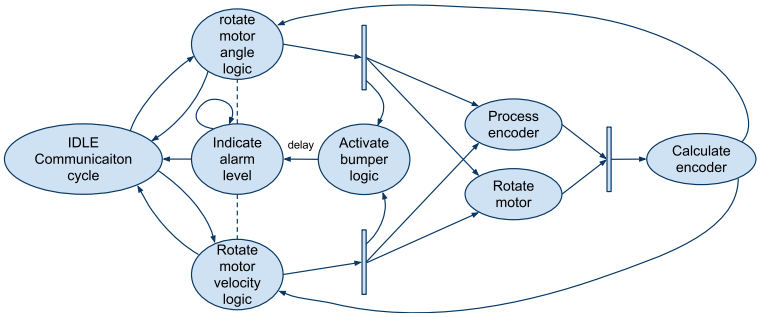

Fig. 5. The process machine for manual control of a motor.

Finally, Fig. 6 shows the processes involved in managing movement of the robot on a flat surface. In this case, the two motors are controlled organically to achieve a common goal: move the robot with constant velocity or move it within a certain distance. By analogy with the previous example, in this case, a separate, simultaneously executable algorithm allows the motors to stop automatically and raises an alarm on dangerous states of the bumper.

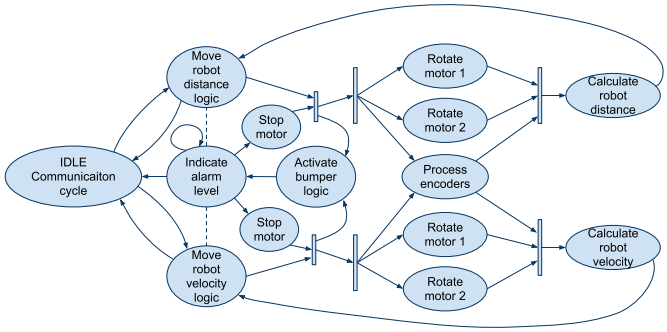

Fig. 6. The process machine for automatic control of the robot's movement.

hus, we formulated both the composition of the system to automate and the content of the processes which are to be automated. Next, let's consider the way these requirements are to be implemented in a distributed control system.

3 The program architecture

As it was briefly mentioned before the distributed control system's core being developed consists of hardware and software parts. The hardware part was described in detail in the previous article [1] and the considered control hardware's configuration was described earlier in this article (cb1 and cb2 boards). From now on we will concentrate on the software part.

Architecture of the software core focuses on such characteristics as:

1. Simplicity and speed of a firmware upgrade for all microcontrollers in a network;
2. Efficiency of the network hardware resources' utilization;
3. Reliability of individual functions through decentralization of the network.

As long as we consider network operation for the core parts an industry networking standard was selected for the low level needs of communication between individual nodes in the system with possible real-time operation. The standard is called CAN (Controller Area Network). This standard [3] does not include the entire stack of the necessary protocols, so it was supplemented by the CAN Kingdom protocol [4], which appeared to be the most flexible implementation of available higher layer protocols for CAN. This protocol allows a system designer to easily specify data flows in an arbitrary way (change software network topology), and also allows a developer to use the whole range of CAN-addresses at his discretion.

To describe the developed approach let's refer to Fig. 7, which shows the levels of interaction in the network in comparison with the standard OSI model of network interactions.

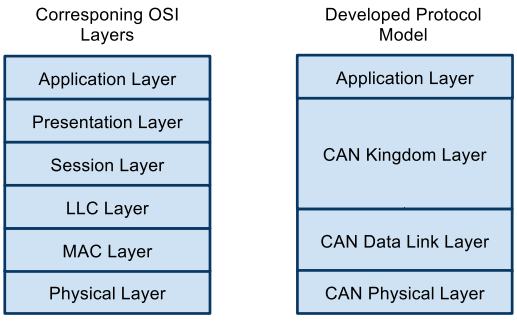

Fig. 7. A comparison of the developed protocol stack and the OSI model.

Generally speaking, CAN Kingdom is not a top-level protocol for the CAN bus, unlike its counterparts - CAN Open, Smart Distributed System, DeviceNet. It does not limit a system developer neither in a field of available CAN-addresses, nor in a logical structure of software parts of a system. However, it introduces some restrictions on the format of packets sent over the network. The key difference of CAN Kingdom from similar protocols is that it defines logical entities with which it is possible in an arbitrary way to organize data streams from one node to another, also allowing distribution of CAN-addresses so as to ensure effective node communication. It is important to note the following CAN Kingdom concept, which in future the authors will expand - "a single point of configuration". This means that all nodes in a system but one should be programmed directly just once and then be able to be integrated into a network, allowing them to read current configuration of it. As a result a system designer has to deal with only one node. That significantly reduces the complexity of the system setup.

Primitive types used in the CAN Kingdom protocol are divided into:

1. Applied types - entities that contain information to be transmitted:
   - Bit - the smallest data cell;
   - Line - a data string of one/several primitive Bits. Maximum size - 8 Bits;
   - Page - a collection of Lines. Maximum size - 8 Lines;
   - Envelope - message IDs (CANID);
   - Letter - a CAN-bus message, a combination of one Page and one Envelope.

   The applied primitives constitute a hierarchy: Letter consists of Envelope and Page, Page consists of several Lines, and Line consists of several Bits.
2. Service types - metadata primitives which are needed for proper transmission/reception of the applied primitives:
   - Form - information describing Page's content;
   - Document - a collection of Forms;
   - List - array of primitives (Bits list, Lines list, Forms list, Documents list ...);
   - Folder - a primitive container that is tied up with one or more Envelopes and one Document.
   - Letter - a CAN-bus message, a combination of one Page and one Envelope.

   The Folder primitive can be compared to a gate through which a node communicates with the network. And associated Document - with rules of entry and exit through these "gates". See Fig. 8 for relations between the primitives.

The CAN Kingdom specification determines a central node in a network, it is called King. This node is responsible for initial setup of other nodes, but before that is possible it is needed to program (a controller's memory or a separate memory chip) all the nodes with the following information:

- A physical address of the node (CANID);

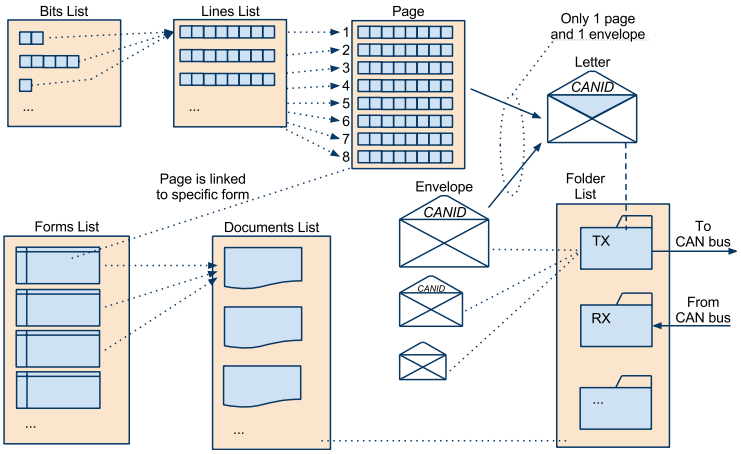

Fig. 8. CAN Kingdom's primitives and relations.

– Data's transfer rate;
– An identification number of the configuration message.

The process of designing a system based on the CAN Kingdom protocol is generally carried out as follows:

1. A system developer is studying modules' documentation. The documentation includes - lists of entities and their descriptions, and algorithms (functions) of the module.
2. A program of a network setup is being developed. The setup is done using 21 predefined configuration messages. During the setup the King node sends a freeze-message to all nodes, after which all of them must stop working on the network and get ready to change the configuration. Next, according to the setup King creates primitives for each node, combining those that are lower in the hierarchy described before in more general entities. Finally it assigns the Folder primitives on each node with identifiers (CANID). Two nodes can communicate with each other only if the Folder primitive on each node has the same CANID and the format of the Document primitive is the same (done in order for nodes with the appropriate Form primitive to be able to recognize received information). After finishing the setup a run message is sent to all nodes, after which they resume normal operation in the new configuration.
3. The resulting network is tested and debugged.

The authors have already begun to implement the CAN Kingdom protocol in the C language. Some of the structures, reflecting the CAN Kingdom primitives are listed below.

```
typedef struct Bits_t { // -- BIT --
```

```
  Direction tr; // direction of transfer ( tx | rx )
  uint8 raw; // raw data
  uint8 sz; // number of meaningful bits in raw data
  uint8 fmt; // format identifier (e.g. m, sm, mm)
} Bits;

typedef struct List_t { // -- LIST --
  Direction tr; // (tx | rx)
  ListContent content;  // bits, lines, forms etc.
  ListId id; // list identifier
  ListCell startCell; // pointer to the list start
} List;

typedef struct Line_t { // -- LINE --
  Direction tr; // ( tx | rx )
  ListCell bitsChain;  //   bit
} Line;

typedef struct Form_t { // -- FORM --
  Direction tr; // (tx | rx)
  uint8 numlines; // number of lines
  ListCell linesChain;
} Form;

typedef struct Doc_t { // -- DOCUMENT --
  Direction tr; // (tx | rx)
  uint8 numForms; // number of associated forms
  ListCell formsChain; // array of forms
} Doc;

typedef struct Envelope_t { // -- ENVELOPE --
  uint8 msb; // most significant byte
  uint8 lsb; // less significant byte
  Tumbler enable; // envelope activation
} Envelope;

typedef struct Folder_t { // -- FOLDER --
  uint8 id; // folder identifier
  ListCell doc; // pointer to document
  Direction tr; // (tx | rx)
  Tumbler enable; // folder activation
  uint8 numEnvs; // number of associated envelopes
  Envelope* envs; // array of envelopes
} Folder;
```

The Page primitive is combined with the Form primitive, and the Letter primitive has no data type, as it is formed only when sending messages and is not intended for storage.

CAN Kingdom's main task is to organize flows of information in a network and to determine the format of transmitted messages. That is not enough for arranging full interaction of distributed nodes of the system. For an efficient and flexible system it is also important to ensure for each node: possibility of algorithms' configuration and possibility of parallel tasks' execution (calculative and applied). The solution to this problem would be the next level of protocol stack that runs on top of the CAN Kingdom protocol.

3.1 The higher level protocol Jet Jerboa

Developing of a high-level application protocol is one of the key tasks in the authors' approach. At this point the protocol is not fully developed, but the basic concept is introduced for its further clarification. The working title of the protocol is "Jet Jerboa". According to this concept a node has several atomic functions. The atomic function is indivisible and runs on a single node. There are also complex functions, which constitute a chain of atomic functions that can run on different nodes. Complex function is the oriented graph in Fig. 9.

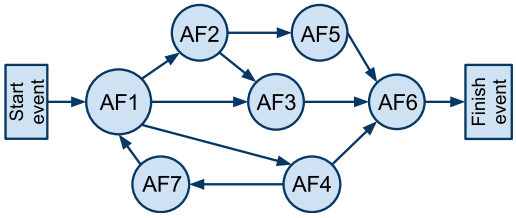

Fig. 9. A complex distributed function.

As one can see, it is possible to form an algorithm of a complex function defining relations between atomic functions, and where possible - execution can be parallelized as atomic functions can run on different nodes. If we introduce the atomic function "iterator" it is also possible to arrange a cyclic operation of some areas of the graph. It will return control to a certain point until some condition is met; otherwise the cycle will be endless.

It is useful to divide atomic functions into two categories:

1. General atomic functions which can be run on any node in a network;
2. Specific atomic functions which can be run only on certain nodes.

All atomic functions have unique identifiers no matter which node they belong to. Start and finish of an atomic function's execution are accompanied by events, which also have unique identifiers within a network. For example, an atomic

function starts execution by the event "at_func_1_req", which was received by a network and finishes with the event "at_func_2_ind", which, depending on configuration, can be transmitted over the network or be a node's purely internal event.

A modifier can be transmitted together with an event. It sets an operation mode for a function. Identifiers of a function's input parameters are defined when configuring a network and are stored in a node's internal table. Before a start of execution it is determined whether a node has needed information, if so, the information is recorded in an appropriate memory cell that then will be used by the function - if not, the node sends a "data request" message over the network.

Finish of a function also constitutes one or more events that can serve as start of other functions' execution. The means to deliver events over a network is the subject of further development.

Memory, as well as computational resources, has distributed nature - each node stores information that can be used by it or by other nodes. Thus it is necessary to distinguish between nodes of information providers and nodes of information consumers. Information providers are usually represented by a variety of sensors. Any node can be an information consumer. In a sensor's measurement a node determines by its internal configurable metadata tables whether it owns the data. It is then stored locally or broadcasted over a network.

It is important to note one feature of the developed architecture - active use of broadcast messages, i.e. messages that are accepted and processed simultaneously by multiple nodes. In some cases this allows to reduce overall network traffic, as well as to increase overall speed of information processing.

3.2 The Jet Jerboa protocol example

To illustrate the Jet Jerboa protocol, let's consider a robot's movement on some route. As before our system consists of 2 components:

- Motion controller (N1)
- Route coordinator (N2)

Table 1. Functions of the N1, N2 nodes.

Function	gId	lId	inParamList	preEvents	postEvents
		N1			
Relative linear movement	1	1	d7,d9	e8	e1
Rotation at angle	2	2	d8,d10	e9	e2
		N2			
Route guidance	3	1	out1,out2	e3..e7,out3	e8,e9
Get the next destination point	4	2	d5..d7	e1,e2	e7..e9

Table 2. Events of the N1, N2 nodes.

Event	gId	lId
N1		
Finish relative movement	1	1
Finish rotation	2	2
Front collision	3	3
No front collision	4	4
Rear collision	5	5
No rear collision	6	4
N2		
Final destination reached	7	1
Start of relative movement	8	2
Start of rotation	9	3

Where:

- gId - a global function identifier, is assigned when configuring a system;
- lId - a local identifier of a function, indicated in module documentation;
- inParamList - an array of input datas identifiers, is initialized on configuration;
- preEvents - an array of identifiers of events causing start of functions;
- postEvents - an array of identifiers of events caused by finish of functions;
- data - data contents;
- out1 .. out3 - external influences and data.

Initially, hardware modules are documented with tables for functions, data and events, but without being tied to a global identifier. Visually a function can be represented as a junction with incoming and outgoing arrows (Fig. 10); where the arrows represent data and events (input or output). Binding of local identifiers (lId) to global (gId) ones occurs during configuration of a system.

Arrows with "e_" prefixes represent events and appear in the network as broadcast packets with the event's global id information. Arrows with "d_" prefixes represent data and can appear in a form of broadcast messages, if the producer of this data is not its user.

When an event message arrives, software modules browse their internal tables of functions searching whether the event is a beginning of a function. If yes, then before starting the function the input data is prepared. The module's local (inner) data table is searched and the identifiers of required data are compared to those stored in the table. If the data is in the inner table - it is written to a startup parameters table of the function without any query on the CAN bus. If the data with the specified identifier is not found in the local data table - a network broadcast request message is dispatched for the data. A node that owns

A. Yudin, M. Semyonov "Distributed Control System for a Mobile Robot: Tasks and Software Architecture" in Research and Education in Robotics - EUROBOT 2011, ser. Communications in Computer and Information Science, vol. 161. International Conference, Prague, Czech Republic, June 15-17, 2011, Proceedings: Springer Berlin Heidelberg, 2011. pp. 321-334.

Table 3. Data of the N1, N2 nodes.

Data	gId	lId	data
			N1
Current coordinate increase	1	1	(Δ1, Δ2)
Current angle increase	2	2	Δ15
Current movement state	3	3	moving
Current bumper state	4	4	front bumping
			N2
Array of route points	5	1	(Δ1, Δ2; Δ0)-(Δ3, Δ4; Δ30)-(Δ5, Δ6; Δ45)
Current point of movement start	6	2	(3,4)
Current destination point	7	3	(5,6)
Current rotate angle	8	4	30
Current movement velocity	9	5	10
Current rotate velocity	10	6	2

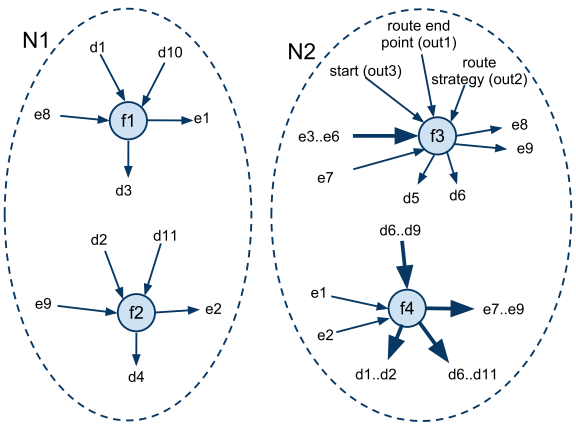

Fig. 10. A graphical representation of functions' binding to data and events.

this data will answer. And finally the obtained data is recorded in the startup parameters table of the function.

Once all parameters are obtained, a function begins execution. The result of it is changed data, which also corresponds to global identifiers that are unique to the entire system. Further on, this data can be written in a local data table if the current node is the owner of the data, or sent over a network with a broadcast message and a data user has to accept it and write it in its data table.

A function execution also results in a list of events which as well as the changed data correspond to global identifiers. When a function finishes operation a network message is sent as an event. All nodes receiving this message check their internal table of functions for required actions and then a process of starting a new function repeats.

Fig. 11 represents a fully connected system of functions.

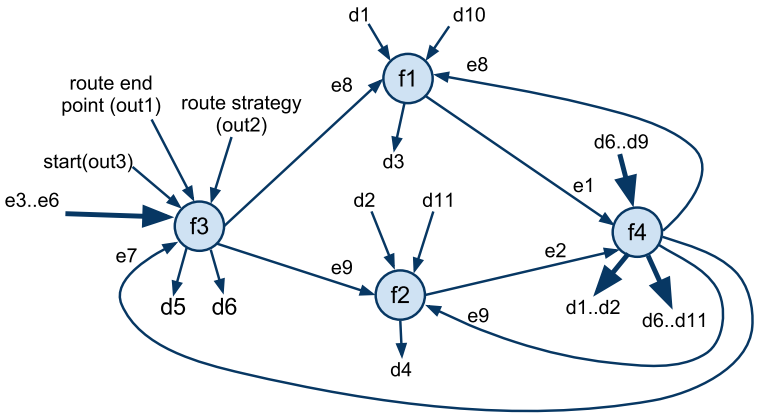

Fig. 11. A system of connected functions.

From the figure one can see that the path e8-e1-e8 and e9-e2-e9 will be repeated for as long as the functions f3 will generate output data d5 different from zero. It should be noted that the proposed architecture solution does not apply for a final decision, but merely illustrates the concept.

4 Conclusions

In this article, the authors tried to summarize the current state of the distributed control system for an autonomous mobile robot. This system was previously presented in a more general form [1].

Detailed examination of tasks put for one of the levels of the system is a good way to better understand the approach being developed. Since the discussed problems of the navigation level are very common in practice of development of mobile robots, this material could be used as a base for future development

of guidelines for the course on competition robotics, read by one of the authors among the first-year students[2].

Developing the previously described approach the authors presented a network model of the system, and also discussed in detail the nuances of the CAN Kingdom protocol used for needs of the system, partly including the software code implementing the protocol for better presentation.

The main point of future development is the Jet Jerboa upper level protocol. This protocol actually has to implement special requirements reasonable for the whole system, which were formulated as follows:

1. Simplicity and speed of a firmware upgrade for all microcontrollers in a network;
2. Efficiency of the network hardware resources' utilization;
3. Reliability of individual functions through decentralization of the network.

An example of the network implementation based on a simple 2 node configuration showed how the network is meant to operate.

In future, the described approach and concept are supposed to form a complete real-time system, the main application of which will be in the field of teaching robotics.

References

1. Vlasov, A., Yudin, A.: Distributed Control System in Mobile Robot Application: General Approach, Realization and Usage. Technical report, 3rd International Conference on Research and Education in Robotics (2010)
2. Eurobot, international robotics contest, `http://www.eurobot.org`
3. CAN Specification, version 2.0. Robert Bosch GmbH, Stuttgart (1991)
4. Fredriksson, L.-B.: CAN Kingdom specification, rev. 3.01. Kvaser AB, Kinnahult, Sweden (1995)
5. Clinger, W.D.: Foundations of Actor Semantics. Technical report, MIT (1981)
6. Robin, M.: Communication and Concurrency. Prentice Hall, International Series in Computer Science (1989)

[2] An optional course on basics in robotics for first year students in Bauman Moscow State Technical University, IU4 department.