Robust robot multitasking programming with ROBOLAB and ULTIMATE ROBOLAB

Claude Baumann, director of the Boarding School “Convict Episcopal de Luxembourg”, Europe

The bulk of the LEGO Mindstorms concept is robot-building and programming. The project is essentially educational and the sets must be seen as sophisticated toys or highbrow didactical tools. In any case, when kids try to get familiar with the stuff, they need generous tutorial support either through direct human guidance or printed and software material. Over the years a huge gallery of exciting robots that have been realized all over the globe complete the official documentation, thanks to many distinguished or anonymous people sharing their efforts in news-groups, papers, conferences and contests.

What is apparent in a large majority of realizations is a concentration on good building practice, whereas clean programming sometimes seems to suffer a bit. Of course, most of the robots are “programmed” in any way, but the efficiency of the programs is rarely comparable to the high quality of the mechanical robot devices they intend to control.

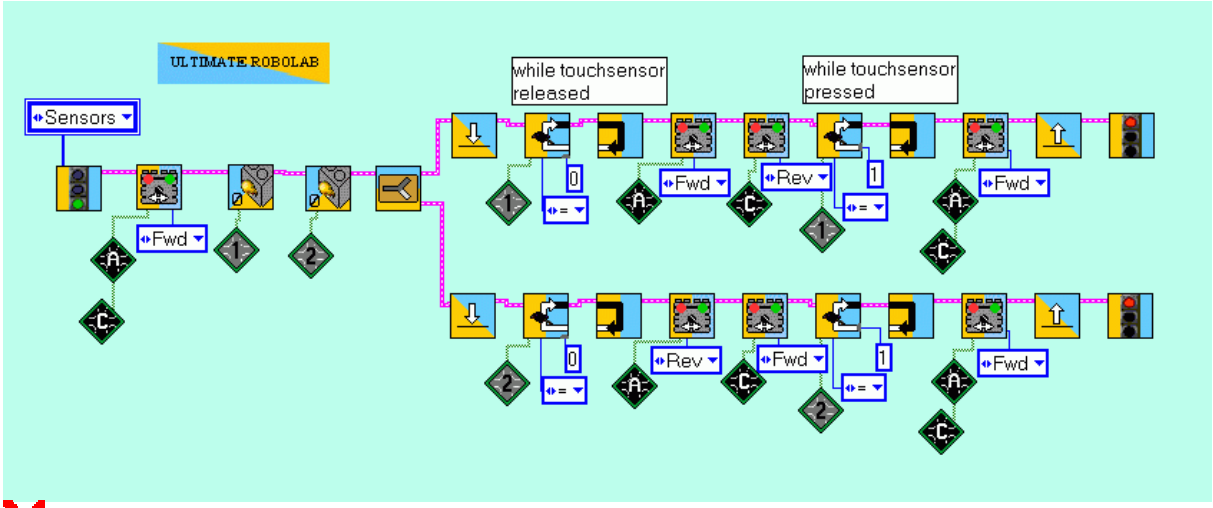
This fact may explain Professor Ole Caprani’s sarcastic exclamation that he made during a LEGO presentation: “*The LEGO advertising phrases: <Easy programming - even for adults> or <Program your robot to do what ever you want> … what a lie !!!*” (Prof Caprani teaches mathematics at Aarhus University Denmark). Far from criticizing LEGO for their marvelous product, this statement must be considered as an honest conclusion from an experienced person who has worked with many groups of children, observing their learning processes with LEGO Mindstorms.

Programming certainly is the complicated part of the Mindstorms project. And even if the initial RIS graphical language and surely the ROBOLAB software have facilitated programming a lot, there remain problems that are hard to understand and no less harder to teach. The truth is, that even simple robots require programs that rapidly evolve far beyond the programming capacities of kids, parents or teachers, often leaving them rather frustrated.

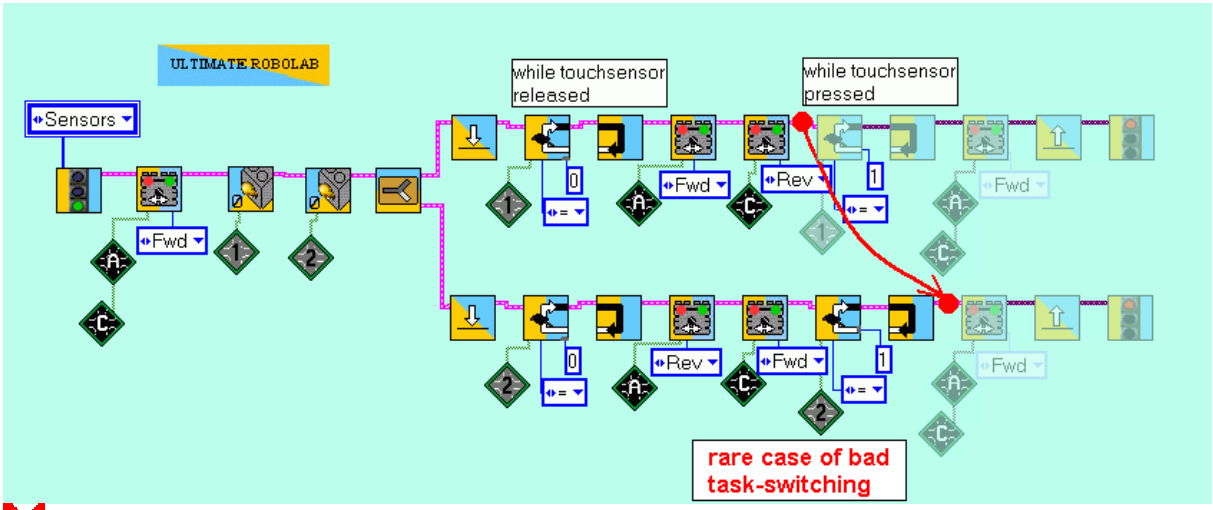
Why does my robot not behave, as I want it to do?

A typical program that caused one of the author’s Mindstorms groups to despair with their robot may be seen in picture 1. The robot was one of those simple dual drive mobile bases equipped with a left and a right bumper. The challenge was to avoid the wall by turning to the right, if the left sensor was triggered and vice versa.

A skilled eye rapidly recognizes the intrinsic problem of the program. But, explaining to youngsters why their robot reacts as it does, is much more complicated than it may seem at first sight. As could have been guessed, the central problem turns around multitasking and may be summarized as the conflicting trial to control resources from more than one process resulting in bad robot states that are recognizable through bizarre behavior like jiggling, if motors are concerned or flickering in the display, or less visible, global variables that take uncontrolled intermediate values.



Picture 1: A simple wall-avoider program causing some trouble. The user has no control over the hidden system task switching.



Picture 2: Seldom possible condition leading to a bad robot state! In this particular case the robot should turn, but the resulting command is going forward. (There exist many subtle conditions that produce such neat effects.)

A closer look to the example in picture 1&2 reveals that the robot could in some rare cases be triggered as follows: suppose touch-sensor 1 being pressed while the multitasking scheduler has allocated the processor to the upper task. The first loop is exited, motor A is set to forward direction, motor C is set to reverse. The robot should start turning to one side. But, imagine that the task-scheduler switches to the lower task exactly at that moment. (Note that the RCX task switching happens approximately at a 1kHz rate). Also suppose that the second task was left at the last task-switching while it was executing the second loop with a bumper 2 release event. Both motors are now set to forward.

Bummer: the robot has taken the wrong decision by choosing a forward direction instead of turning away!

The risk that this bad state or a similar one may be reached is very small, but the rare erroneous reactions certainly cause headaches to the robot-designers, especially to kids that don't get the problem, because they have so many difficulties to understand the multitasking processing. In a professional environment, any program reaching

such undesired and uncontrolled states is absolutely unacceptable. Advanced ROBOLAB programmers should be aware of the difficulties and learn how to avoid them.

The example presents another, more obvious problem: if the wall-avoiding robot gets stuck in a corner, it will alternately turn left and right without ever finding its way out. This proves that the wall avoiding strategy defined by “if hit on left bumper, turn to the right and vice-versa” is not sufficient to solve the challenge in an all around satisfying way. If the robot reaches the 90° corner with an incidence angle of about 45°, both bumpers will be hit nearly synchronously, but the robot will first react on one sensor, then on the second one. In fact there are more sensor states that should generate a robot reaction than actually anticipated by the program:

1. No bumper hit
2. Left bumper hit
3. Right bumper hit
4. Both bumpers hit

5… A kind of double click : left-right, or vice-versa; either with intermediate release or not.

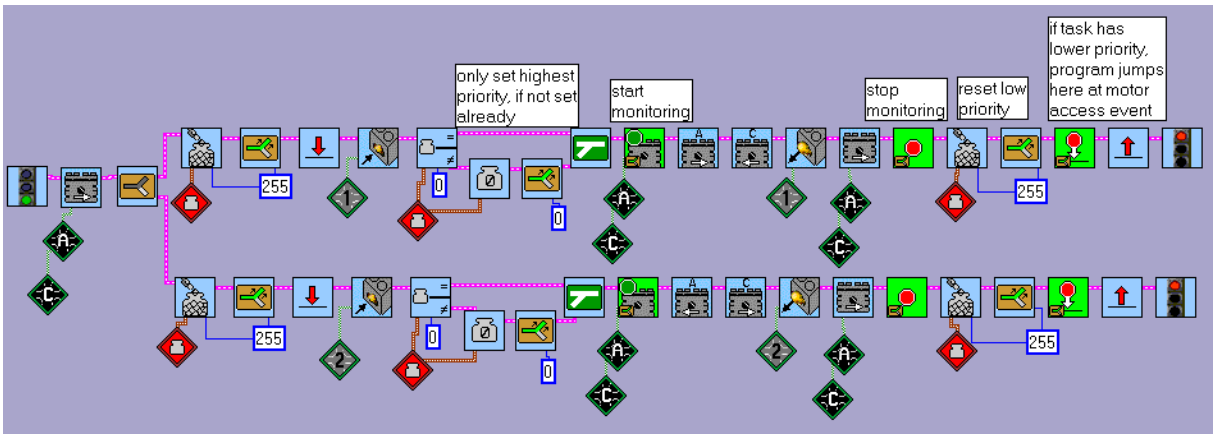
It can hardly be imagined, that the robot could develop a controlled reaction to states that it doesn’t even recognize.

Avoiding parallelism conflicts

There are many ways to overcome problems linked to the multitasking activity. Most of them obey to the rule of the growing complexity, but some are quite easy to manipulate. In the following we will very pragmatically explore three important classes of solutions on the base of practical examples:

- output-access-control through task-hierarchy and output-monitoring
- control of critical program sections through active task-management
- resource arbitration management

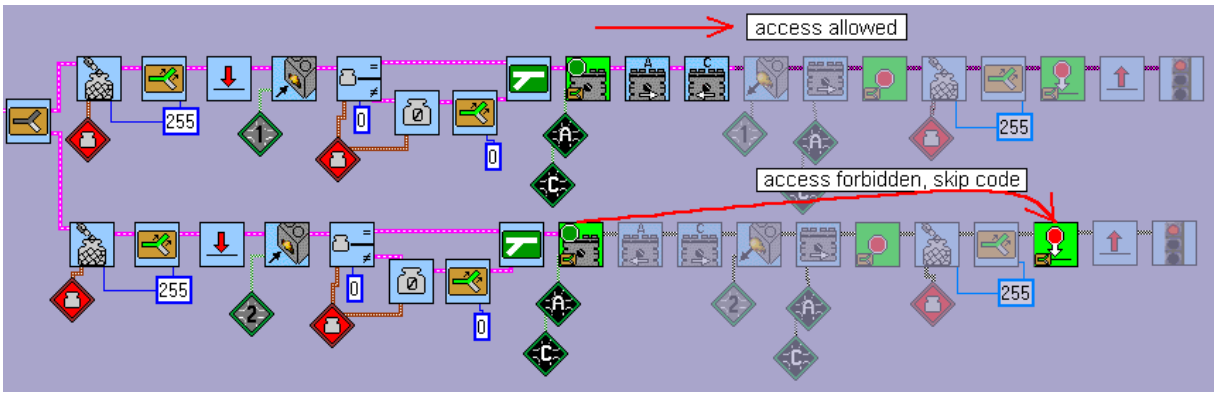
A) Advanced access-control with standard ROBOLAB



Picture 3: A good method to control motor access. Only one task at a time may have the highest priority (0) and change motor states. Any other task that tries to get control over the motors jumps to the special land.

The idea behind the output access control is to place a survey monitor between the attempt to control outputs and the effective output commands. Only the highest priority task is allowed to access the outputs. By this way, any other trial fails and the commands are ignored. The great quality of this approach is that non-authorized tasks are only affected in their behavior, if they really try to access the output resource. In other words, tasks may continue doing non-critical jobs, enhancing the program efficiency.

If a hierarchy is engaged in the control system, which is the case with the LEGO access control, the programmer must provide an astute strategy to allow two or more tasks of the same priority to keep equality in accessing critical sections. The trick that has been applied here is to increase the priority as soon as the chronologically first task enters the critical area. Thus the program obeys to the “First Come First Served” strategy, even if the hierarchy normally would have delivered a different result.



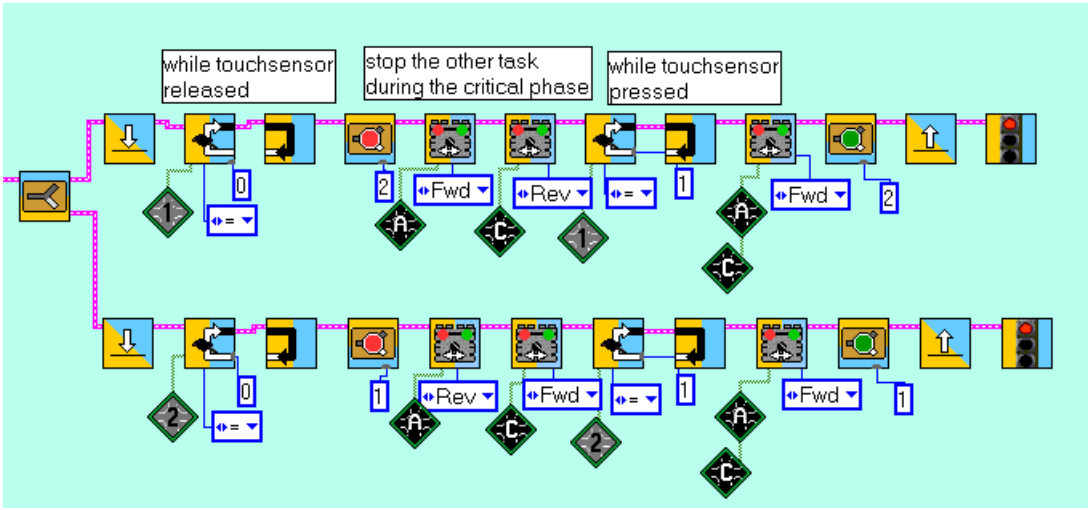
Picture 4: The high priority task may freely walk through the code, but the lower priority task is forced to skip the critical code.

The disadvantage of the output access control structure is that it rapidly evolves to complex programs. When adding it to the firmware, LEGO was aware of this problem. The engineers describe the RCX-firmware 2.0 in the paper *Command Overview* released with the beta-version, writing to the topic: *"When event monitoring is used in combination with access control across several tasks, a complicated regime must be implemented ..."* Follows a two-dimensional [7 x 5] array of events, states and reactions. Being based on firmware 2.0, ROBOLAB (>= v. 2.5) implements the features of event-monitoring and access-control.

Because of the growing complexity if more tasks and other resource types are engaged, Ultimate ROBOLAB has no built-in access-control, but allows a few exciting and uncomplicated ways to efficiently overcome the multitasking problem that shall be exposed here.

B) Active access-management with Ultimate ROBOLAB

1. Radical safe methods



Picture 5: This radical solution tries to stop the opposite process, whenever the current process enters a critical section that should not be interrupted.

The multitasking access problem can be drastically avoided, if the opposite task isn't allowed to enter the critical program part that may be the origin of the conflict. This can easily be done as shown on picture 5. If the first task reaches the "stop task" instruction, the other task is halted. This means that the task-scheduler isn't allowed anymore to switch to that task. Now the remaining task may freely control the motors without any trouble. Once the touch-sensor is released and the motors are reset to the default direction, the inhibited task is reactivated. Note that both tasks are equal from the hierarchic point of view, each one may equitably reach the higher control level.

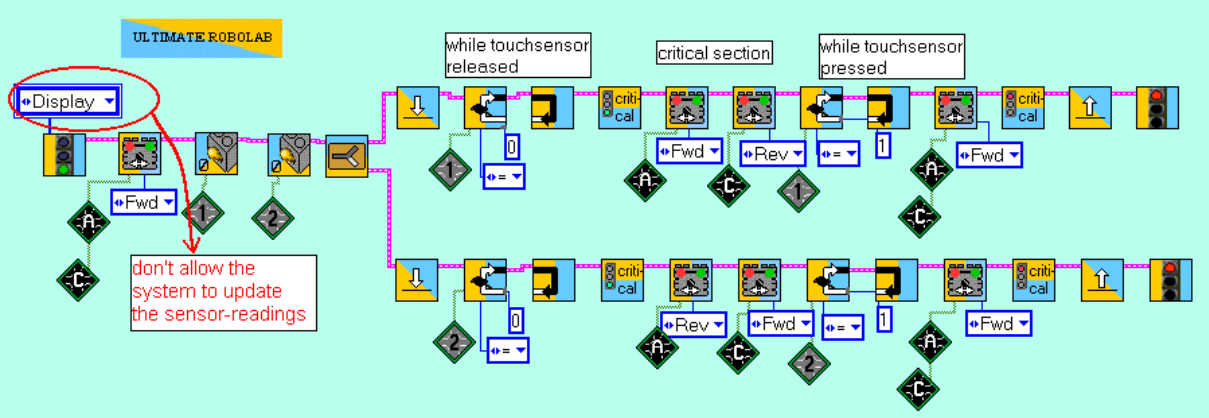
The main disadvantage of this method is that tasks may be stopped, even if they didn't try to enter the critical section. This can lead to serious inefficiency with more complex programs, because processes could be interrupted during vital robot-functions. Furthermore, the method should not be used with a larger number of tasks for the reason that each task with critical sections needs to be stopped. Thus the program will obviously be overcharged with task-management instructions.

Ultimate ROBOLAB allows even more radical methods. Notifying the task-scheduler that a certain program part is "critical" blocks any try of exiting the active task until this state is cleared. The consequence is that the particular important system task running normally in the background and controlling things like the display, sound or sensor reading is blocked too. Note that for several reasons the Ultimate ROBOLAB kernel only distinguishes three different task states : ready, critical and inhibited. A "ready" task may be freely run by the task-scheduler. A "critical" task may be entered, but not exited. An "inhibited" task may be exited, but not entered. This simple task-management, a trade-off to speed and memory requirements, allows very clean and robust RCX programs.

The most radical method of controlling the access to critical sections is the complete disabling of the hardware interrupt that clocks the entire RCX. In that case, any other RCX function different from the active task is inhibited, including the task-scheduler and the input/output control devices. If this method was applied to the described example, neither the sensors nor the motors were actualized anymore and the program would not run at all.

Radical methods should only be used shortly and with extreme care. In general however the task-stop-go, the “critical” method and the interrupt disabling-enabling strategy are very easy to implement and require only a few lines of code. One must not forget that sophisticated control-systems are quite costly in terms of memory space and computing time. It is up to the programmer to proceed to a skilled weighting. If the critical section is extremely short, it is obvious that the access controlling code should not grow over a certain size.

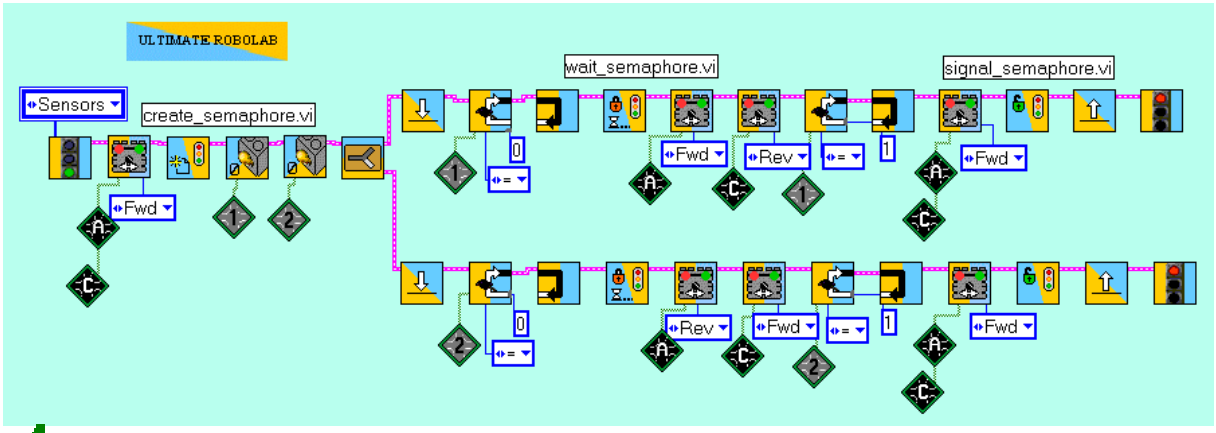
Picture 6 shows a valuable but discussable implementation of the “critical” solution. If a task enters the critical section, the system background task is halted together with any other task except the one that entered the critical section. Practically it is important therefore to unselect the system sensor update. Ultimate automatically transfers the sensor updating into the active task. If not well configured in the “Begin” icon the program would hang up, because the RCX were no longer able to detect any touch sensor change and would be eternally lost in one of the sensor-loops. The added system option “Display” allows the system to normally update the RCX display and show the tiny running man. But, if one of the bumpers is triggered, the man will stop running for the duration of the critical phase. This demonstrates the radicalism of the method: the background-system really is blocked.



Picture 6: During the execution of this most radical method, the program halts the task-scheduler, affecting all non-critical tasks, including the basic system task that normally updates the sensors. (Ultimate ROBOLAB transfers the sensor-refreshing into the sensor-loops.)

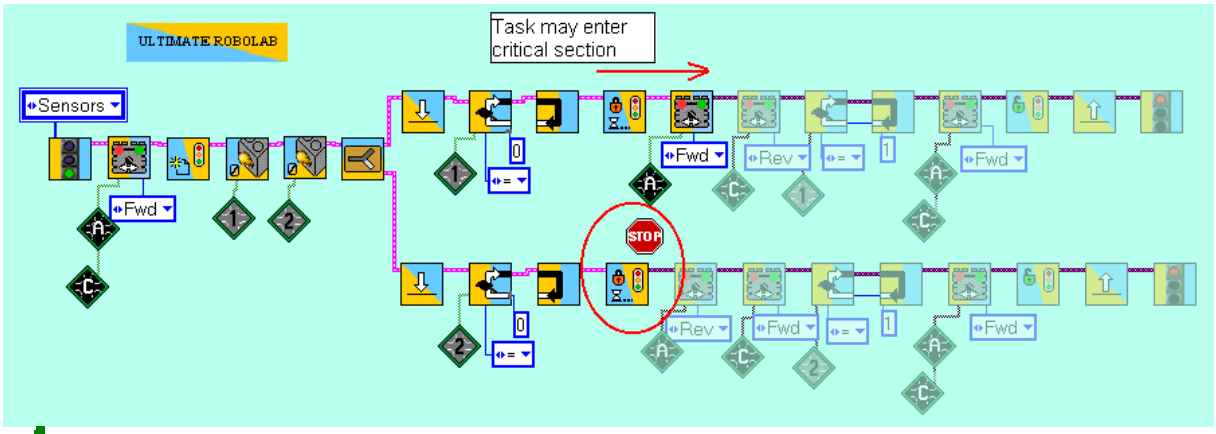
2. A perfect method based on messages

Ultimate ROBOLAB has an implementation of so-called semaphores that allow tasks to communicate with each other. The Dutch mathematician E. Dijkstra, one of the most brilliant early computer scientists, invented this program construct in 1965. The idea roughly is that each time a task tries to enter a critical code section, it checks, if this is allowed. If not so, its task identification-number is added to a queue and it is immediately stopped. Anytime a task is leaving a critical section, the scheduler is told that the first task in the queue -if ever there is one- may enter the critical region. This very efficient system overcomes the problems that appeared with the radical methods. Tasks are only stopped, if they try to enter an occupied critical section. A direct consequence is that non-critical code is not affected by the semaphore and the system keeps on running in a stable manner.



Picture 7: A semaphore is an extremely powerful and easy to implement system that allows communication between tasks.

Besides the enormous flexibility of the semaphore-structure, this astute programming technique is characterized by simplicity, shortness in code and extendibility, when working with hierarchally equal tasks. Adding more tasks doesn't increase the program complexity from the access control point of view. And above all, a program can be set up with more than one semaphore. The method has also its limits, when a hierarchical distinction of tasks is desired.



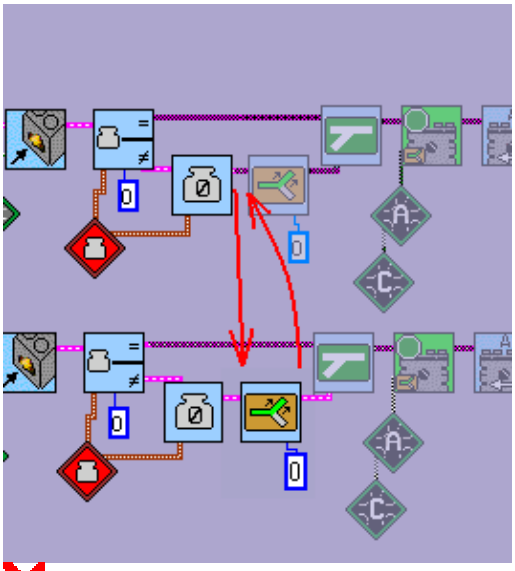
Picture 8: Semaphores follow the strategy "First Come First Served".

3. Important note

Embedded applications like RCX programs that are running under multitasking conditions may in absolutely rare cases produce undesired results that hardly can be explained from a higher programming language point of view. These reactions are the effects of very low level bugs or program incoherencies that in most of the cases are localized in the multitasking management. First of all, one must point out that the whole method palette from access control to semaphores represent themselves program code that is submitted to the same access problems. The communication either with the access monitor or the task-scheduler may well comprise critical sections, because global variables are engaged, that need to be written and read from different places. For example, the semaphore queue is a data structure that has two pointers one for writing, the other for reading. They must be correctly adjusted at the right moment, before or after any data-transfer. During any operation it must be made sure that no other task is disturbing the action. Second, multitasking processing is a very complex mechanism that needs a lot of system and task-specific

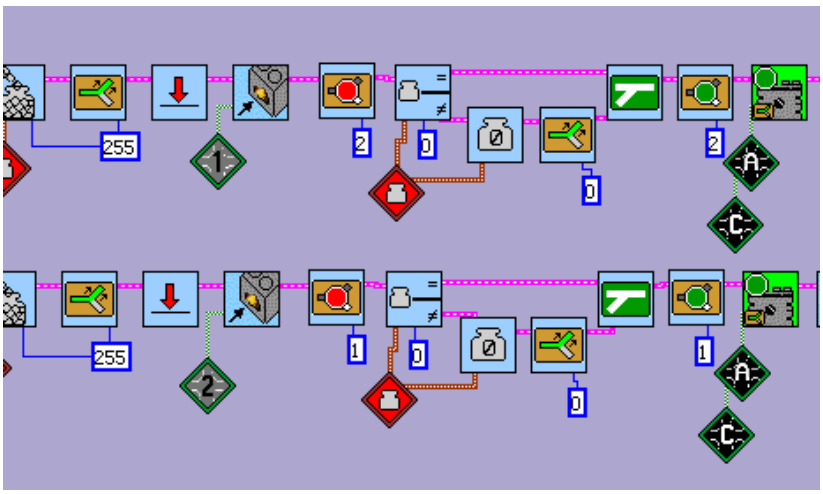
data to be moved around in the memory. Aware of Murphy's laws, one must be skeptic enough to conclude that the more complex a program-code works, the more sensible it will react to the unexpected. "Bluescreens" are the proof to this assertion.

Perhaps the attentive reader (listener) noticed that the ROBOLAB access control solution in picture 3&4 have been marked with an orange tilt sign. In fact the program carries an inherent problem that may illustrate what has been said in the previous paragraph. The task-switching could have taken place at the very wrong moment with the result of both task priorities are set to highest. This example demonstrates well that the parallelism collusion problem always appears, if there is an attempt to write to the same variable from more than one process.



Picture 9: Problematic super rare condition: the task-switching leaves the upper task just before having set the priority, then switches to the lower task. Now the priority of this task is set to highest (0). The task-scheduler switches back to the upper task, continuing where the process was left → the priority is also set to highest !!! Conflict !

Since standard ROBOLAB has no built-in semaphores, there is no other choice to avoid the collusion than through disabling the opposite task for a short moment. So, we add the radical start-stop mechanism … and …



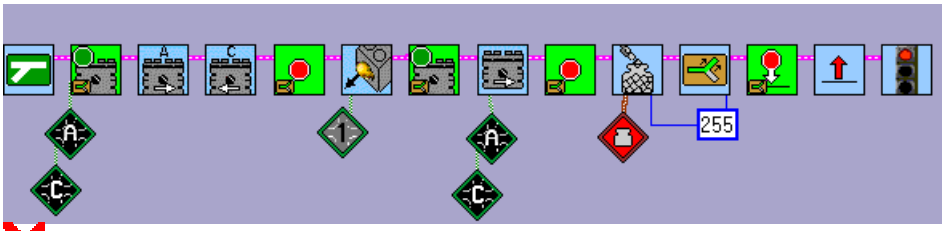
? … the program does not work anymore, because of a LEGO firmware bug !!! (Picture 10)

C) Resource arbitration methods

Each one of the described control systems presents its advantages and disadvantages. One important criterion that should be observed, while deciding which method shall be implemented for a given robot challenge is the question, if a hierarchical task-structure is desired or if the tasks should have equal priority. This question is not so trivial as it seems, since even simple robots may need both structures at the same time.

A more valuable program for our wall-avoider should be able to detect corners and react accordingly. If no special hardware “corner detector” is added to the robot - whatever this may be - the software must be set up to solve the challenge. There could be an additional task that is starting a timer, if one bumper is hit, and see, if the second bumper is also hit within a certain delay. A positive result should then override any other robot reaction. In this case there would be three tasks, with the particularity that two of them have the same lower priority and the additional task occupies the higher one.

The problem is even emphasized, since the higher priority task must have the faculty to suppress any lower level behavior. This is particular difficult, because all the methods that have been described so far include the wait-for-bumper-release loop inside of the critical section. One bad solution could be to only embrace the direct motor commands between the critical section delimiter to give each task more freedom.



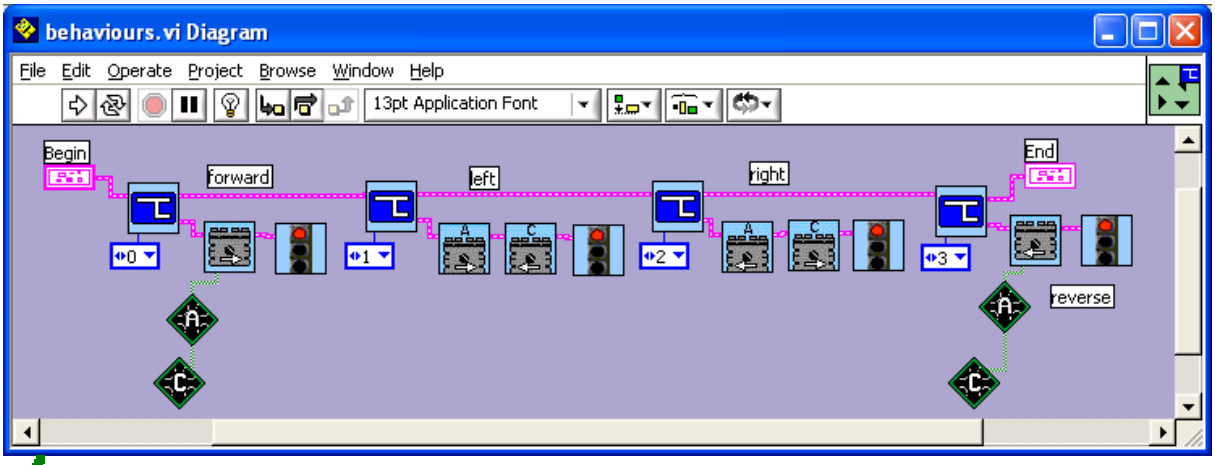
Picture 11: More task liberty also means, more problems. The difficulties of the initial example (picture 1&2) reappear. (It is a good exercise to find out why !)

Solving the parallelism turns out to be much more complicated than perhaps expected. The necessity of robustness, correctness and efficiency asks careful programming. However the hardest requirement to a valuable solution is to keep the programming structure simple enough that inexperienced programmers can work with it.

Starting from these challenging settings, it is now time to switch over to a complete change of the paradigm : instead of protecting the resources from being accessed from more than one process, the resource control is allocated to one only task that could be called the resource manager or the arbitrate. If other tasks wish to produce a certain robot behavior that involves the resources, these tasks must notify a request to the arbitration task that decides according to priority criteria, which one of the requests will be executed.

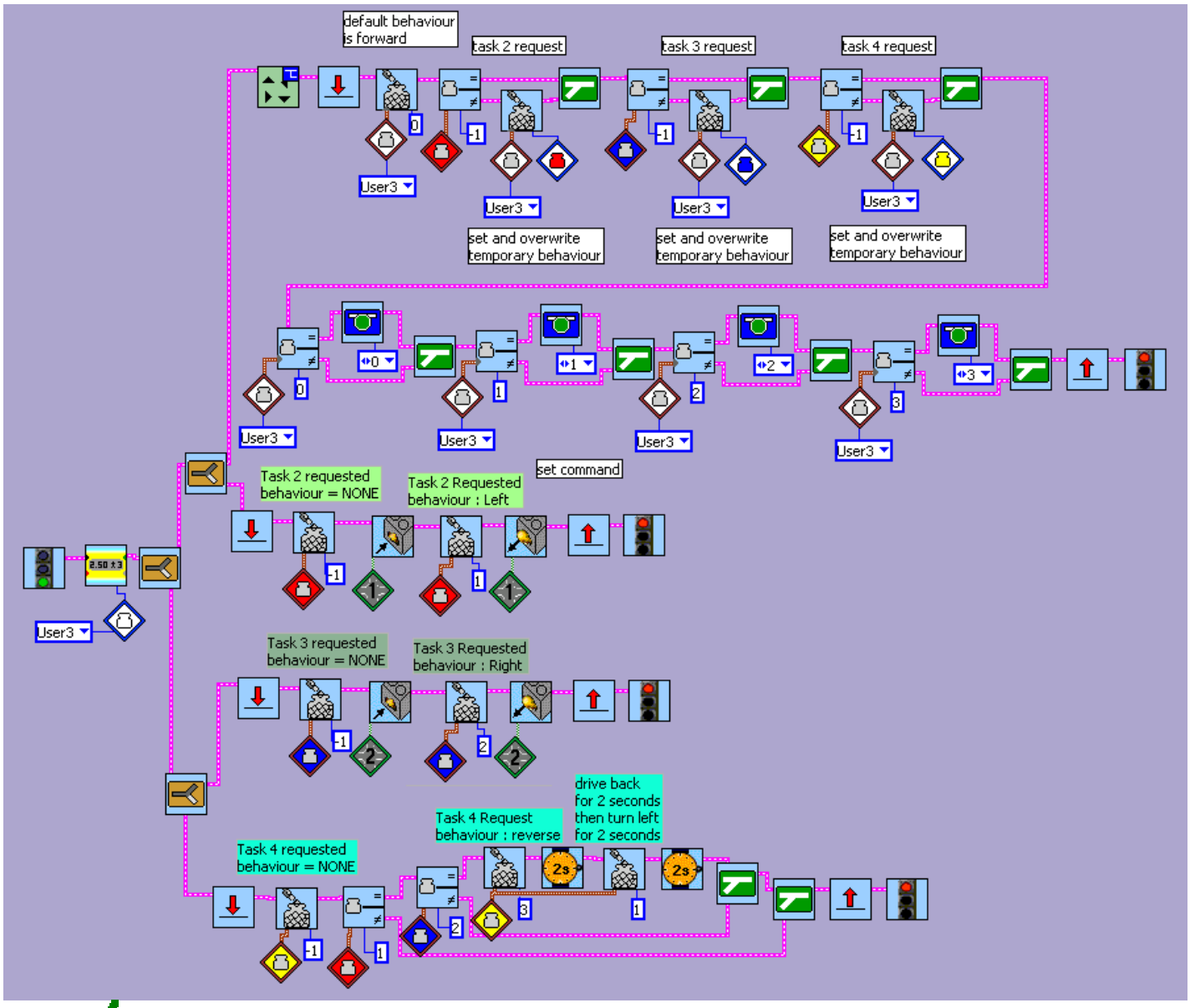
There exist various incarnations of this programming concept. One of the most popular and powerful techniques is the “subsumption architecture” that has been developed by Rodney Brooks at MIT in the late 1980s. Two main clever mechanisms characterize this structure : first, all the desired robot behavior-tasks run simultaneously. To avoid any conflicts, none of them has direct access to the outputs and higher priority behaviors replace or subsume the lower level ones. Since there always exists only one highest level behavior, this particular robot-state effectively controls the robot and its resources. Transitions from state to state are executed on the base of sensor-values or messages, but they are decided by the arbitration task alone. The second mechanism concerns the communication of the behavior requests which is done through a mixture of temporary local and permanent global variables. By this way, the communication is reduced to simple robot-memory read- and write actions with the particularity that communication variables are only written to by one only task, if there is certainty about the robot behavior. Computer memory represents the best protected resource, since access to it is managed by hardware means. No processor allows writing and reading to the same memory-location at the same time. Thus, intermediate critical sections are atomized to nearly nothing.

All this sounds awfully complicated, but with some minor efforts the subsumption architecture may be implemented into the RCX. Picture 13 shows the program of an improved wall-avoider with corner detection. To keep the overloaded diagram readable, both sensor tasks have different priorities. But, where is the hierarchy hidden? The trick here is that the main motor control container “User3” - only used in the resource manager task - is first set to the default value causing the robot to advance, if no other requests is notified, then set to a value received from bumper 1 task, then from bumper 2 task and finally from the corner detection task (both requests set). But container “User3” is only changed, if the request-value is different from -1, which means that there is in fact a request. This successive overwriting produces a hierarchy given by the order red, blue yellow. In any case there is a clear command value in “User3” (0..3) at the end of the sequence. Each value stands for a certain robot-behavior and the behavior subroutines are called accordingly.



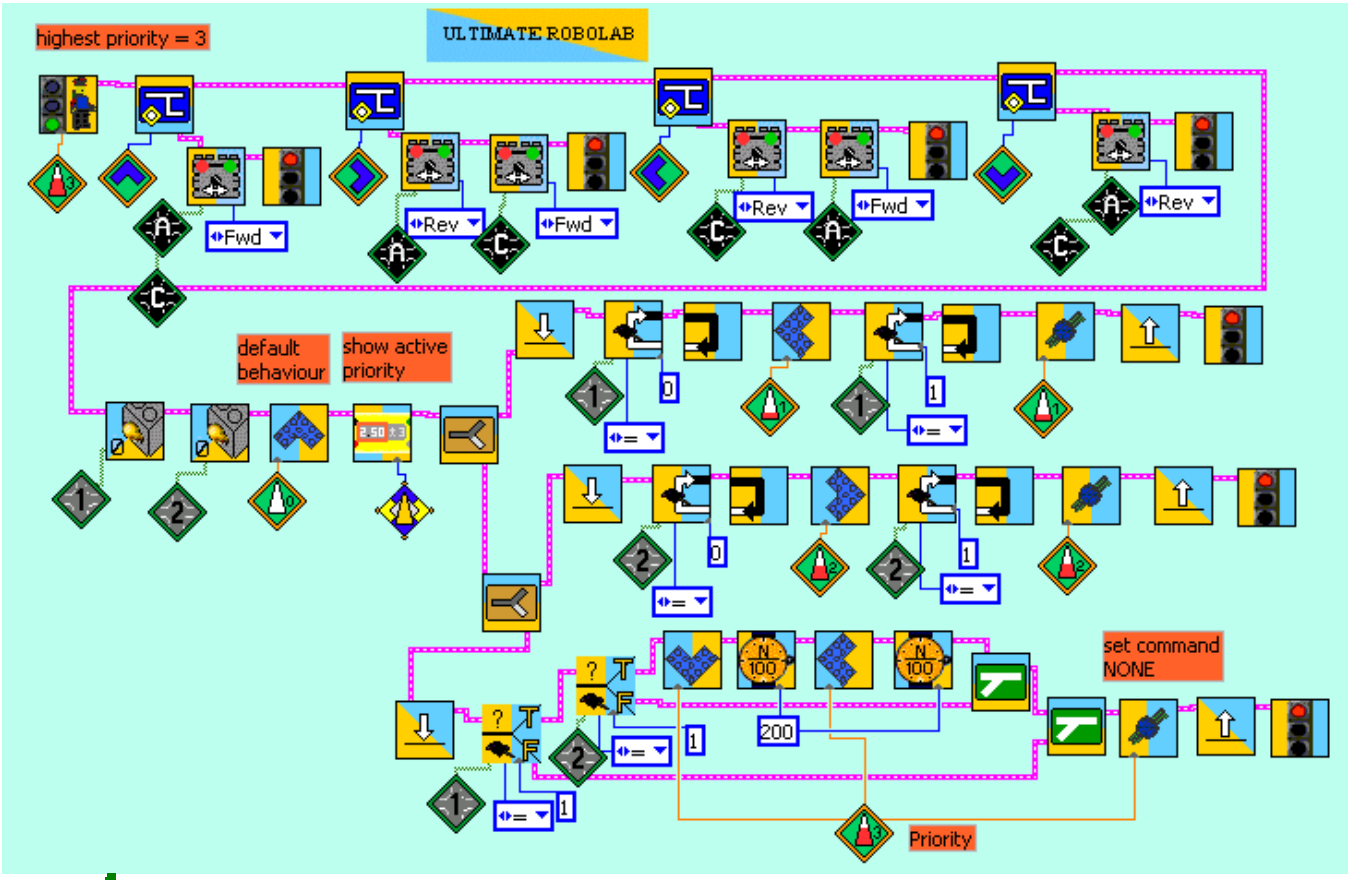
Picture 12: The behavior subroutines are defined in a self-made sub.vi

It is a bit more difficult to manage equal priorities with this system. But for the present robot this actually doesn’t matter, because the both-sensors-pressed-event generates an own behavior. If you run the current program, you’ll notice a certain display-flickering that is caused by the task-specific variable “User3” changes.



Picture 13: A better wall avoider based on the subsumption architecture. The upper task represents the resource control manager.

With Ultimate ROBOLAB things become much easier, since the software has a complete built-in behavior control module that is optimized in both memory economy and reaction speed and obeys to the subsumption architecture. Besides this it is very easy to manipulate. With Ultimate the wall-avoider program would look have the following aspect :



Picture 14: Ultimate ROBOLAB allows very easy manipulation with behavior controlled robots. Thanks to the built-in subsumption architecture module, there are no parallelism collusions anymore.

To allow better hierarchic control, the priority may be used either static or dynamic. It is also possible to communicate messages from tasks to the arbitrate task, which is completely integrated into the Begin icon. The program can be easily improved to have both bumper tasks react with the same priority, if a semaphore is added.

Conclusion

This non-exhaustive excurse into the complex topic of parallel processing, exploring the dangers of access conflicts, if there is an attempt of possessing data or resources from different tasks, may have helped to explain bizarre robot behaviors that could have appeared during robot courses or training sessions and that didn't find any satisfying explanation. The presented methods to avoid collusion problems certainly are not in the reach of children, except for the subsumption architecture implementation. With the necessary preparation to have strong ROBOLAB icons that replace complicated program-code, this programming construct may be successfully used with all levels of Mindstorms groups.
