

Everything You Always Wanted to Know
About Interactive C But Were Too Lazy
To Read The 6.270 Manual.

By Kevin Harrelson

Machine Intelligence Lab
University of Florida
Spring, 1995

Introduction

This document is not meant to be an all-inclusive reference manual for IC. Instead, it is meant to show a couple of the features that might not be obvious or intuitive. This document assumes that the reader has had some basic experience programming in C.

Deviations from standard C & Cautions

Because IC is a special purpose language, certain features have been left out in order to make the code manageable. IC does not recognize case and switch. In addition, there is no provision for the comma operator. If you do not know what these are, don't worry about it because you couldn't use them anyway.

Multi-dimensional arrays are not supported. Please refer to *Everything You Always Wanted to Know About Behaviors But Were Afraid To Ask*, by Kevin Harreslon, for details on overcoming this limitation.

Character type variables can be accessed ONLY as arrays. **Do not try to access a single element of a character type array.** This will lead to curious errors and a corrupted array.

Another thing not to do involves the hardware. IC uses an interrupt, presumably for the multitasking control. This interrupt causes port A4 to generate pulses that appear random. Obviously, this pin will be unusable for anything.

Multitasking in IC.

IC has the ability to perform multitasking. This means that the processor will be able to virtually run several programs at once. While it may not seem possible for a single processor to do this, it is accomplished through time slicing.

The information presented here is meant to be a quick guide. More information and details can be found in *The 6.270 Robot Builder's Guide*, by Fred Martin. Please refer to section 7.9 for more details on this (you should have read this already, *right?*).

Time Slicing

Time slicing works on a simple principle. Since the processor works extremely rapidly, it can appear to do several things at once by spending a little bit of time on one process, then spending another little bit of time on the next process, and so on. In this manner, each process will get some processor time. Because the each process gets a little "slice" of time, this process is known as time slicing.

For this scheme to work, it is necessary to have a process manager that can interrupt one process when its allotted time is up, and pass control to the next process. Therefore, this procedure is totally transparent to the processes being run. A program cannot tell when, or even if it has been interrupted. Also, all processes are completely independent. They can only communicate with each other by using global variables.

Controlling Processes

Initially, there is only one process running on IC. In effect, this is the process that is responsible for communicating with the host computer. Whenever a command is typed directly at the IC prompt, this is the process that will execute it. IC controls all processes in terms of TICKS. A single TICK is one millisecond.

The `start_process(function() ,TICKS, STACK-SIZE);` is the basic function that tells the process manager to add a new process. The `function()` can be any IC function or statement. The `TICKS` is the number of ticks to be allocated for this process. The `STACK-SIZE` determines how much “scratch-pad” memory is allocated for the process. `TICKS` will default to 5 milliseconds, and `STACK-SIZE` will default to 255. For most applications, the defaults will be adequate. If a process is running too slow relative to other processes then `TICKS` can be increased. It will seldom be necessary to decrease `TICKS`, because of the `defer() ;` statement, which will be discussed later. The default `STACK-SIZE` should be increased if IC reports an error concerning a process running out of stack space. It should only be decreased if the programs are large enough so that the processor is running out of memory, which is unlikely. The `start_process(function());` statement will return a number, which is known as the process identification number, or `PID`. It should also be noted that whatever value is returned by the function that is started will be lost.

The `kill_process(PID);` statement will kill a process. The `PID` is the same one that was returned when the process was created. As a practical matter, it is unlikely that a program will use this statement. All processes can end themselves by issuing a `return;` statement.

There are two additional commands that can be used on other processes. They will only work from the IC prompt, and *cannot* be placed in programs. The `ps` command will list all processes, along with the `PID`, `TICKS`, and other information. The `kill_all` command will stop all processes.

There are two commands that a process can use to control itself. By using the `defer() ;` function, a process gives up the rest of its time slice for that turn. However, the process will get its full time on its next turn. The `hog_processor() ;` function will add 256 milliseconds to its current turn. Because this function can easily cause one process to take over the machine forever, it should only be used when necessary, such as when executing time critical code.

Programming Tips

In order to maintain control of the robot, the `main()` process should simply initialize any needed variables, start other processes, and exit. This should be done because, if `main()` is started from the IC prompt, the prompt will not return until `main()` is done executing. If it goes into a loop and actually starts processing data for the robot, then the IC prompt will not reappear unless `main()` is interrupted. By having `main()` set everything up and then exit, interactive debugging will be possible. See the *Debugging in IC* section for further details on this.

Use the `defer()` function. When programming behaviors, the sensor data should be placed in global variable, which are updated by a sensor servicing routine. Obviously, if a behavior runs and produces output, it should then defer. If it does run again in the same time slice, the input will be the same because the sensor servicing routine has not yet had a chance to run, and the output will be the same. Thus, a short behavior that does not defer is wasting valuable processor time.

Debugging in IC

While writing and debugging programs in IC, you may often wonder “Just what is this @*\$! program doing?” One of the nicest features about IC is that it behaves like an interpreted language. While this makes the language relatively slow, it does make it much easier to debug.

One form of debugging might involve using the `serial.c`, but this will require exiting IC, and loading a terminal program. It is also impossible alter any variables while doing this.

There are two things that suggest that there is an easy way. First, any global variables can be displayed in IC while the machine is running. Second, when an array is displayed in such a manner, *every* element of the array is displayed. The procedure would then seem to be simple. Just define a global array, and a global pointer. A routine could then be called that will dump a number into the array, and increment the pointer. All a program has to do is call the debugging function, and pass a number. The number could be a sensor output, a behavior output, a behavior number, or any other output.

In order for this trick to work, however, you must have control over the robot while it is running. To do this, have your routine, `main()`, simply initialize variables and call other routines, and then exit. When `main()` exits, you will be able to communicate with IC, while all of your behaviors are running in the background. Please refer to the *Multitasking in IC* section for further details.