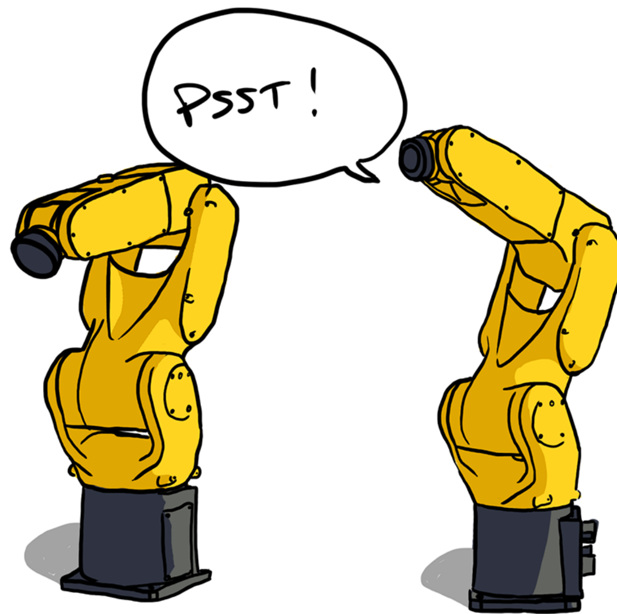
ONE ROBOTICS COMPANY

Robot Whispering

The unofficial guide to programming FANUC robots



Jay Strybis

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Robot Whispering

The Unofficial Guide to Programming FANUC Robots

by [Jay Strybis] of [ONE Robotics Company]

v1.1

©2018 ONE Robotics Company. All rights reserved.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Table of Contents

1. Introduction
2. A Very Simple Program: Hello, World
3. How TP Programs Work
4. Creating a Program with Motion
5. `STEP` Mode and Moving Backwards
6. Control Flow
7. Numeric Registers and Assignment
8. Multiple Displays
9. Modifying Positions (Direct-Entry and Touching Up)
10. Using I/O
11. Subroutines
12. Frames of Reference (`UFRAME`s and `UTOOL`s)
13. Motion Options
14. Joint and Linear Motion
15. Payloads
16. Conditionals
17. User Alarms
18. Simulating Inputs
19. Remarks
20. `WAIT`-statements

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

21. The `END`-statement
22. Getting the Robot Home Safely
23. Reference Positions
24. Cycle Stop and the `ABORT` statement
25. **Appendix A:** Mapping I/O

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Introduction

Learning how to program FANUC Robots was, for me, a process of trial and error. I was lucky enough to start my career at FANUC America's headquarters back in 2008. As an engineer in the Material Handling group, I was given the opportunity to try (and often break) many things with the comfortable safety net of helpful, more-experienced colleagues and near-unlimited access to robots and software.

I left in 2011 to pursue a side-venture in web development but quickly started getting phone calls to help people with their robots. As it turns out, I actually enjoy programming robots more than web servers, so I've focused on that almost exclusively ever since. As a consultant, I'm paid to get things working quickly the first time. Luckily I learned from plenty of mistakes while working for FANUC.

With this book I've attempted to distill these years of trial and error down into the core tools, techniques and best-practices I use on every job. When you've finished reading, you'll know how to compose programs that work every time, avoiding common anti-patterns and pitfalls I've seen other programmers often make. With a firm foundational-level understanding of the FANUC robot programming environment and a lot of practice solving common problems, you'll find yourself readily quipped to handle the majority of applications you'll come across.

I encourage you to actually write and test every code example as they are presented. It may not be obvious at first while reading, but I will sometimes do things "the hard way" before showing a better way of accomplishing the same task. My hope is that experiencing the pain will help the lesson to sink in.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

NOTE: *I've tried to provide not only the information you need to get going but also some background on why things work the way they do. These details are often included in* ***NOTE:*** *blocks like this one.*

A Very Simple Program: Hello, World.

It wouldn't be a book on programming unless it started with a "Hello, World" example. Feel free to skip this short section if you're already familiar with creating new programs, adding instructions and running programs from the teach pendant.

Here's the finished product:

```
MESSAGE[Hello, World.] ;
```

For the purposes of this book, I will assume that you have a teach pendant (physical or virtual via ROBOGUIDE) handy, but you could also write the instructions by hand.

Creating a Program

First we have to enable the teach pendant. See that `ON/OFF` switch in the top left corner? Make sure it's switched to `ON`. This makes you, the holder of the teach pendant, the sole person in control of the robot.

Now that your teach pendant is `ON`, let's create a new program. Press the `SELECT` button to get to the robot's list of programs. Next press the `F2 CREATE` *softkey* to create a new one.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

> ***NOTE:*** *The* `F1` *through* `F5` *keys are called softkeys. The function of each button changes depending on the current screen context. When we're on the* `SELECT` *screen's list of programs,* `F2` *creates a new program. It likely does something completely different (or nothing at all) on other screens.*

You are now asked to give your program a name. Use the `F1`-`F5` softkeys to name your program. The process for typing strings with the teach pendant is like an old cell phone: press the softkey multiple times to get the next letter (e.g. press `F1` once for A, twice for B, three times for C, etc.) Use the arrow keys and backspace button if you have to. It's a huge pain, but you'll get better at it with practice.

> ***NOTE:*** *If you're using an iPendant, you can also bring up a full onscreen keyboard via* `Options/Keybd > KEYBOARD`.

Press `ENTER` when you're done, and you'll be presented with an empty editor with which to write your first TP program.

Adding an Instruction

Notice that your softkeys have changed. We're in an editor context now, so their functions are now relevant to making program edits.

> ***NOTE:*** *If you see the little right arrow > to the right of the* `F5` *softkey label, that means you can press the* `NEXT` *button to see more softkeys for the current screen.*

Press `NEXT` until you see the `INST` softkey above `F1`. A menu pops up from which you'll be able to navigate and choose which instruction to add to your program and

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

edit. Press `0` (or use the arrow keys) to cycle through the menus until you see the Miscellaneous option. Choose that one by pressing `ENTER` (or the number associated with that option) and then navigate to the Message instruction. Select that instruction (again, with `ENTER` or the number associated with it) and you should see an empty `MESSAGE[]` instruction on the first line of your program.

Use the arrow keys to move the cursor around your program. Move it until it's between the two brackets of the `MESSAGE` instruction and then press `ENTER`. Now, just like naming your program, type out the message, "Hello, World!" and press `ENTER` again to finish editing.

Your program should look like this:

```
MESSAGE[Hello, World.] ;
```

Congratulations! You just wrote your first TP program.

Running TP Programs

The key switch on your robot controller determines whether the robot is in teach mode or AUTO. When the robot is flipped over to T1, the mechanical unit will only move at the command of the enabled teach pendant, and the robot Tool Center Point (TCP) is limited to a maximum speed of `250mm/sec`.

When in AUTO, the robot is usually controlled by some external device like a PLC. At minimum there should be a safety fence to keep people safe from the robot, and it's usually a good idea to have Dual Check Safety (DCS) setup, but that's outside the scope of this book. Let's focus on teach for now.

Running in teach requires several conditions to be met:

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

1. The teach pendant is enabled
2. The controller T1/AUTO select switch has to be on T1 (or T2 if you have it)
3. The deadman switch has to be pressed half-way (don't worry about this in ROBOGUIDE)
4. No faults are present (press `RESET` to clear faults)

If all of the above are true and your editor still shows your program active, press the `SHIFT` and `FWD` keys at the same time. You may not have realized it, but your program just ran. Press `SHIFT+FWD` again, and it will run again.

"Where did my program go?"

The `MESSAGE` instruction brings up the `USER` screen. That's where those messages get printed. You can bring this screen up whenever you want by pressing `MENU` then selecting the `USER` option.

Depending on how many times you ran your program, you should see "Hello, World." written once for each time. You actually don't have to be on the editor screen to run your program. Press `SHIFT+FWD` again (making sure there are no faults first, of course) and you should see the friendly robot greet the world again.

Why Did We Do That?

This first program is obviously trivial, but we did set the foundation for what you'll need to do on every job going forward:

1. Creating programs
2. Adding instructions
3. Editing instructions

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

4. Running programs

Before moving on to more complex programs, let's learn a bit more about how the robot actually interprets and runs them.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

How TP Programs Work

Think of a TP program as a list of instructions. When you run a program, the controller simply starts at the top and executes each instruction in order, one line at a time until it reaches the end.

You've seen the `MESSAGE[]` instruction which writes to the `User` screen. Other instructions tell the robot to move, mark locations, jump to marked locations, talk to other machines, and some are just comments for your reference.

By the time you're done with this book, you'll know a lot about the set of instructions necessary to accomplish most tasks.

Creating a Program with Motion

Go ahead and create a new program. (**REMEMBER:** hit `SELECT` for the list of programs, then `F2` to `CREATE` a new one.) Once you have your blank editor, we're going to add your first motion statement.

Motion Instructions and Recording Points

Motion instructions are special. They even get their own softkey so you don't have to dig through the `INST` menu. Press the `NEXT` key until you see the `POINT` softkey above `F1`, then press `F1`.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

A list of options for what type of motion statement you want to add pops up. Don't worry about the list itself right now (we'll get to those later) and instead use a shortcut to insert the first option. Press `SHIFT` and `F1` at the same time. You should see a motion statement like the one below:

```
J @P[1] 100% FINE ;
```

Let's break this instruction down:

1. The `J` indicates that this is a *Joint* motion statement. The robot can move from point A to point B a few different ways, Joint being one of them, but we'll cover that in detail later.
2. The second part is the *destination* of the motion statement: `P[1]` or position 1. The position stores the location of the robot TCP the moment you recorded it locally within the current program. (**NOTE:** See the `@`-symbol to the left of the destination? That means the robot is currently at that point.)
3. The third part is the *speed* of this motion statement: `100%`, as fast as it can go. (**NOTE:** You could change the units from `%` to `sec` or `msec` here, but let's use `%` for now.)
4. Last is the *termination type* of this motion statement. The termination type basically dictates how the robot should behave as it reaches the destination.

Termination Types

There are two main termination types: `FINE` and Continuous (`CNT`). `FINE` moves cause the robot to come to a complete stop at the destination before moving on. Continu-

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

ous terminations accept an argument between 0 and 100 that specifies how much the robot should decelerate when approaching the destination. `CNT0` causes the most deceleration (similar to `FINE`) while `CNT100` is the fastest, allowing the robot to breeze by the point, rounding corners as necessary.

Let's change the termination type on our motion statement from `FINE` to `CNT0` just to get the hang of it. Move the cursor with the arrow keys until the `FINE` termination type is highlighted. Press `F4` for `CHOICE`, choose the `Cnt` option and enter a value of `0`. Go ahead and change the segment speed too if you want.

Test things out by running the program. Hold the deadman switch in, clear your faults, press `SHIFT+FWD` and hold your breath as the robot… just sits there and does absolutely nothing.

Sorry for the let-down, but this is working precisely as intended. When a motion statement is executed, the robot will move from wherever it is to the programmed destination. Since the robot's already at `P[1]`, there's no reason for any motion.

Jog the robot up a bit and then run your program. You should see the robot move down, back to your taught point. Jog the robot left and run your program. The robot moves right, back to the taught point.

> ***NOTE:*** *The robot does all the hard work of figuring out how to get from point A to point B. Can you imagine having to do this all yourself? This makes our job much easier, but it can also be dangerous. Be very careful to manage how your motion instructions get executed so as not to inadvertently drive the robot through a wall, itself, the floor, etc.*

Let's record another point so the robot actually does some work. Make sure the cursor is on the line below the first motion statement (use the arrow keys if you have to), jog

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

the robot somewhere else and press SHIFT+F1 to record another point and add another motion statement. Your program probably looks something like this:

```
J  P[1] 100% CNT0 ;
J @P[2] 100% FINE ;
```

> ***REMEMBER:*** *the @-symbol indicates you're currently at* P[2].

Run the program again. The first line moves the robot from its current position to P[1]. The second line moves it back to P[2].

STEP Mode and Moving Backwards

Sometimes it's helpful to cursor through your program and execute just one line at a time. Press the STEP key and you'll see the step-mode indicator turn on at the top of the screen. Pressing SHIFT+FWD now executes just a single instruction, pausing the task after it's done.

Step through both lines of code, then press SHIFT+BWD. You'll see the robot move back to P[1] as the cursor moves back up to line 1. You may find it useful to step back and forth through your motion statements to fine-tune positions, speeds and CNT-values.

Control Flow

Motion is a big part of robot programming, but making decisions based on the current state of the system is the other part. This is mostly done with looping and branch-

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

ing control structures. We'll start with the most basic building blocks of TP control flow: label definitions (`LBL`s) and unconditional jump-statements (`JMP LBL[]`).

Labels allow you to mark locations in your programs. You can then use corresponding `JMP LBL[]` instructions to "jump" back (or forward) to those marked locations.

Let's add a label to the top of our program. We don't want to overwrite the motion statement, so let's first add a blank line directly above it. Cursor up to the first line, then hit the `NEXT` key until the softkey above `F5` says `EDCMD`. Press `F5` and then choose `INSERT`. The editor asks you how many new lines you would like to insert. Press `1` (or just leave it blank) and then press `ENTER` to insert one new line above the current cursor position.

The `INST` softkey should be above `F1`. If not, hit `NEXT` until it is. Press `F1` and choose the `JMP/LBL` option. Choose the `LBL` instruction from the list to insert your first label. The label is inserted, and your cursor will be between the two brackets, waiting for you to enter a numeric identifier for it.

TP labels are uniquely identified (within the current program's scope) by positive integers. You don't *have* to define them in any sort of order, but it may help you keep your program organized if you define them with some sort of structure in mind (e.g. ascending order).

> ***TIP:*** *Don't go overboard with labels… the fewer the better.*

> ***NOTE:*** *you can add an optional comment to a label by pressing* `ENTER` *on the numeric identifier (e.g.* `LBL[1:top]`*)*

Try stepping through your program again. What happens when you execute the line 1 label definition? That's right: nothing. This is one of those cases where an instruction is

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

really just there for reference. Let's give some purpose to our label by adding a `JMP` instruction to the bottom of the program.

Cursor down to the very last line where it says `[End]` and add a `JMP` instruction from the `JMP/LBL` menu. Your instruction will be inserted, and your cursor will be sitting on the label index waiting for you to enter a number. Enter whatever you defined earlier so that the task will jump back up to your label when the `JMP`-statement is executed.

> ***NOTE:*** *the editor will NOT validate the label you reference in your* `JMP`*-statement until it's executed. Attempting to jump to a nonexistent label will throw an error.*

Test your program and notice how it now runs continuously. When the `JMP` instruction is executed, the cursor jumps back to the top label and starts over again.

> ***NOTE:*** *Most people frown upon using unconditional jumps like these. Here's a rather famous article from Edgar Dijkstra, one of the founding fathers of computer science, criticizing "Go To Statements" in 1968. It's very easy to create "spaghetti code" that is difficult to follow. TP's numeric identifiers for labels exacerbate the problem by conveying zero meaning to the instruction. What is* `LBL[14]` *for again? I have no idea.*
>
> *Most modern programming languages have syntax that make manual label definitions and jumps largely unnecessary, but they are a necessary evil with TP unfortunately.* ***Do yourself a favor and use them as sparingly as possible.***

Your program should look something like this:

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

```
LBL[1] ;
J P[1] 100% CNT0 ;
J P[2] 100% CNT0 ;
JMP LBL[1] ;
```

The robot loops between `P[1]` and `P[2]` for as long as you let it. Let's count the repetitions to introduce you to a couple of other instructions.

Numeric Registers and Assignment

By default the robot gives you 200 numeric registers with which to store numbers: `1`, `3.14`, `42`, etc. You can see this list of numbers by hitting the `DATA` button (right next to the `SELECT` and `EDIT` keys.)

> ***NOTE:*** *If you don't see a list of numeric registers when you hit* `DATA`*, you can hit* `F1` *for* `TYPE` *and choose Numeric Registers. Many screens give you an* `F1` *softkey for* `TYPE`*, so keep an eye out for this if you ever get an unexpected screen or just want to explore the teach pendant.*

It's good practice to label your registers as you use them. Do this by pressing `ENTER` on the blank space next to the numeric ID for `R[1]`. Let's call this first numeric register "counter."

Go back to the editor and insert a new line above your label. We're going to initialize our counter to 0 here. Add an assignment instruction by choosing `F1` for `INST`, then choose Registers and finally the **…=…** option.

The instruction is placed into your program in an unfinished state. It will want you to choose the receiver of the assignment. Choose `R[]` and enter the index of `1` be-

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

tween the brackets. Next it wants you to enter a value. Choose Constant and enter `0` as the value. Your finished assignment should look like this: `R[1:counter]=0 ;`

> ***NOTE:*** *If the comment is not visible, it may be that you need to toggle the comment visibility via the* `EDCMD > Comment` *command.*

You can probably guess how we're going to count how many loops the robot does: another assignment statement. The only difference is we are going to assign the value of a simple addition expression to our numeric register instead of a constant.

Insert a new line above your `JMP`-statement and then insert another Register instruction but this time choose the **…=…+…** option. Fill in `R[1]` for the receiver and the first operand. Use a constant of `1` for the last operand of the addition expression. Your program should now look like this:

```
R[1:counter]=0 ;
LBL[1] ;
J P[1] 100% CNT0 ;
J P[2] 100% CNT0 ;
R[1:counter]=R[1:counter]+1 ;
JMP LBL[1] ;
```

Run your program and allow the robot to move back and forth a few times and then go to the `DATA` screen to check the value of `R[1]`. It should equal how many times you let the robot move to `P[2]` from `P[1]`.

Multiple Displays

It would be nice if we could see our register incrementing while the program is running. While you could run your program with the `DATA` screen pulled up, you might

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

want to keep an eye on the program too. Lucky for us the iPendant supports multiple displays.

Press `SHIFT+DISP` and choose Double. Bam! Two screens. If your iPendant has a touch-screen, you can simply touch one of the panes to bring it into focus. If not, you'll have to use the `DISP` button to toggle through them. Pull up your editor in the left window and put the `DATA` screen on the right. You're quickly becoming a teach pendant power-user.

> ***NOTE:*** *I had you pull up the editor on the left side because this window is the only place you can run programs from. You'll get an error message when attempting to* `SHIFT+FWD` *with any other window in focus.*

Modifying Positions (Direct-Entry and Touching Up)

What if`P[2]` isn't exactly where you want it? How do you update that position?

1. You can change the position coordinates directly.
2. You can "touch up" the point by jogging the robot to where you would like it and recording the new adjusted position.

Let's start with option 1 and change it directly.

Cursor over until you've highlighted the 2 between `P[2]`'s brackets. You should see `Position` as the `F5` softkey (**NOTE:** you may have to press `Next` to bring it up.) Hit `F5`, and now you can see the actual cartesian coordinates of `P[2]`. You can alter

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

these here, but be careful: there's no validation here. Mistype a number and you may crash the robot or make the position unreachable.

> ***TIP:*** *I typically only use direct-entry to round numbers out (e.g. change* `499.721` *to* `500.0` *or* `179.98` *to* `180.0`*). I may also quickly eyeball the orientation of the wrist before directly entering the pitch, yaw and roll values to square things up perfectly.*

Option 2 might be a better option for us. Jog the robot to a new position and then move the cursor so that it's on the line with the motion statement to `P[2]`. You should see the `TOUCHUP` softkey appear above `F5`. Press `F5`.

Did I get you? Nothing should have happened. In order to prevent people from accidentally touching up positions, FANUC requires you to press `SHIFT` and `F5` at the same time.

Using I/O

Your robot's not going to be very useful if all it can do is move from point to point. It will probably have an end-of-arm tool (EOAT) attached to the faceplate that will allow it to pick and place or perform some other function.

The EOAT is usually attached to the EE (for End Effector) connector on your mechanical unit. The EE connector has pins for your Robot I/O (denoted by `RI[]` and `RO[]`). For this example, let's assume robot output 1 `RO[1]` turns on a vacuum gripper, and `RO[2]` turns it off. Let's add a comment to these two output signals before adding any code to our program.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

NOTE: *You don't have to use the EE connector for your EOAT, but it's probably the most convenient. For more information on mapping I/O, check out* ***Appendix A*** *at the end.*

If you still have two screens open, switch focus to the screen on the right by tapping the pane or pressing the `DISP` key. Press `I/O` then `F1` for `TYPE` and select `Robot`. Press `F4 DETAIL` (press `Next` if necessary) on the first output and type "Vacuum On" into the Comment field. Press `Prev` when you're done to get back to the list of outputs. Label the second one "Vacuum Off." Move the pendant's focus back to the left screen and make sure the editor is up by hitting the `EDIT` key.

NOTE: *"Do we really need two signals?" Maybe not… though this is actually quite common with pneumatics. Bare with me for now. We'll get to a useful feature for handling this pair of signals soon enough.*

Let's imagine our robot is picking something up at `P[1]` and dropping it off at `P[2]`. We'll need to add two new lines: one to turn the vacuum on at `P[1]`, and another one to turn it off at `P[2]`. (**REMEMBER:** Use the `F5 EDCMD` softkey to insert new lines.)

Cursor up to the first line and add an I/O instruction. Choose the `RO[]=...` option. Specify `1` as the index and choose `ON` as the value. Do something similar for the other new line but turn `RO[2:Vacuum Off] ON` instead.

Your program should now look like this:

```
R[1:counter]=0 ;
LBL[1] ;
J P[1] 100% CNT0 ;
RO[1:Vacuum ON]=ON ;
J P[2] 100% CNT0 ;
RO[2:Vacuum OFF]=ON ;
R[1:counter]=R[1:counter]+1 ;
```


Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

```
JMP LBL[1] ;
```

Run your program while keeping an eye on the Robot Output status screen in the right pane.

Notice a problem? While both outputs were probably `OFF` to begin with, they're both `ON` all the time after just one cycle. You *could* add a couple more statements to turn the bits `OFF` as needed, but there's actually a built-in feature for this common instance when outputs need to complement each other.

Complementary I/O

Go back to the I/O status screen and select `F4 DETAIL` on `RO[1]`. One of the items on your detail screen is Complementary. Set this to `TRUE` with the `F4` softkey. You will need to cycle power to the controller for this to take effect.

> ***NOTE:*** *You can cycle power from the pendant with* `FCTN > Cycle Power`. *This is a pretty common task, so you may get the* `FCTN > 0 > 8` *key sequence memorized quickly.*

With this feature enabled, the controller automatically turns `RO[1] OFF` whenever `RO[2]` is `ON` and vice-versa. This lets us simply worry about turning these `ON` in our program; the controller will take care of the `OFF` states and making sure they're never both `ON` or both `OFF` at the same time.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Subroutines

You could keep programming like this: adding more and more instructions to your main routine as needed (and many people do), but I don't recommend it.

In my opinion, you will be much more productive (and less error-prone) if you compose your robot's behavior with small chunks of very specific functionality. Let's clean up our main program by extracting some of what we've written into their own subroutines.

Create a new program (**REMEMBER:** `SELECT > CREATE`) and name it "GRIP." Now let's `COPY` and `PASTE` our "gripping" I/O statement into this program. Go back to the `SELECT` screen and press `ENTER` to select and edit your main routine. Cursor down to the line that reads `RO[1:Vacuum ON]=ON`. Press `F5` for `EDCMD` and then choose `Copy/Cut`. Press `F2` to `SELECT` just the current line and then press `COPY`. That instruction now resides in the editor's paste buffer.

Now `SELECT` the `GRIP` program, hit `EDCMD` then `Copy/Cut` again and press `PASTE` followed by `LOGIC`. You should see the `RO[1:Vacuum ON]=ON` line of code appear on line 1.

> ***NOTE:*** *You could have pressed* `POSID` *or* `POSITION` *with the same effect. These special options are for pasting motion statements. We'll cover them later.*

Create another program called "UNGRIP" and do the same thing with the `RO[2:Vacuum OFF]=ON` instruction.

We now have two new very specific, semantically-named routines that we can use as necessary to "grip" and "ungrip" parts whenever we need to. If our gripper I/O re-

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

quirements ever change, we will only need to update these programs instead of all the places the I/O-statements might have been used otherwise.

> ***NOTE:*** *I used the names "GRIP" and "UNGRIP" vs. something like "VACUUM_ON" and "VACUUM_OFF" intentionally. While you may guess that turning the vacuum on "grips" a part, other types of grippers may not be so clear. Grippers that engage the outer-diameter of a part need to close in order to grip. The opposite is true if the inner-diameter is engaged. Closing the gripper ungrips the part while opening grips it.* ***Naming matters.***

Let's go back to our main routine and use them. Cursor up to each of the I/O statements and overwrite it by adding a new instruction. Choose the `CALL` statement from the `CALL` instruction menu, then choose the appropriate subroutine from the list. Your program should now look like this:

```
R[1:counter]=0 ;
LBL[1] ;
J P[1] 100% CNT0 ;
CALL GRIP ;
J P[2] 100% CNT0 ;
CALL UNGRIP ;
R[1:counter]=R[1:counter]+1 ;
JMP LBL[1] ;
```

If you `STEP` through your program, you'll notice that your task actually switches from your main routine to the `GRIP` and `UNGRIP` subroutines when they are called. You fall back to the calling routine once each subroutine ends.

If you want some more practice, try to refactor the counter logic into its own set of subroutines.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

NOTE: *You can read more about this on my blog where I highly recommend writing small, focused programs. Maybe it's the fact that you can generally only see 20 lines at a time on the pendant. Maybe it's just good practice (don't repeat yourself, single responsibility principle, etc.). Whatever the case, I think you'll find, as I do, that small programs will make you much more productive in the long run.*

Frames of Reference

Think back to your early math classes where you had to plot points on a 2D X-Y graph. Points like (2,3) or (-5,4) were relative to the intersection of the X- and Y- axes. Those coordinate pairs would be meaningless without some reference frame. Rotate or move your graph, and the points move with it. FANUC robot positions work the same way.

UFRAMEs

User Frames (or `UFRAME`s) define custom frames of reference for your positions. By default, everything is relative to the robot's `WORLD` user frame, `UFRAME[0]`. The actual `WORLD` Origin varies depending on the robot model, but it's generally right in the middle of the J1-axis at the height of the J2 servo motors. All of your positions will be relative to this point by default.

While you may use `WORLD`-relative positions for some things, you'll probably want to use your own frames of reference for most positions.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Why? Here's an illustration: let's say you just spent the afternoon fine-tuning a path around some part that sits on a fixture. Just as you're getting ready to leave for the day, a coworker walks by and casually mentions that the fixture needs to move over a bit to make room for something.

Defined a frame for that fixture? No problem: you'll just reteach the frame when the fixture is moved, and the points will move along with it.

Recorded everything in `WORLD`? Take a deep breath, realize you'll have to do a lot of work over again and remember to teach a user frame first this time so you won't have to teach all those points a third time.

UTOOLs

The other thing to consider with motion is *what part of the robot* we are moving to reach any given position. By default, this reference point on the physical robot (your TCP) is right in the middle of the faceplate (the end of the robot wrist.) The robot arm will manipulate itself such that *this point* meets any taught positions.

Using Tool Frames is a good idea for many of the same reasons that using User Frames is. For example: if your robot has a suction cup that's 200mm below the face-plate, doesn't it make more sense to record points relative to the cup itself than the faceplate? What if the gripper gets shortened by 50mm? Would you rather update 100 points or a single `UTOOL` value?

Whether you're using custom `UFRAME`s and `UTOOL`s or not, it's always a good idea to explicitly set them at the top of any program with motion. If you don't, you can easily run into conflicts between what references were active when you recorded your positions vs. what's currently active.

Let's review these important concepts:

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

- The `WORLD` user frame (`UFRAME[0]`) represents a point in space that's almost always in the center of the robot base at the height of the J2-axis.
- The default zero `UTOOL` is at the center of the robot faceplate.
- All positions are relative to some frame of reference.
- The robot moves such that its current `UTOOL` reaches the target point.

Setting the UFRAME and UTOOL

Following in the tradition of creating short, specific programs, let's create a couple that we can use to set our `UFRAME` and `UTOOL`.

Create a new program called `UFRAME_PICK` and add the `UFRAME_NUM=...` instruction from the Offset/Frames instruction menu. We'll use 1 for our imaginary pick fixture. Your program should look like this:

```
UFRAME_NUM=1 ;
```

Create another program called `UFRAME_PLACE` and use `UFRAME[2]` for that one: `UFRAME_NUM=2 ;`.

Create another program named `UTOOL_GRIPPER` and add the `UTOOL_NUM=...` instruction from the same Offset/Frames instruction menu you used previously. Set a value of 1 that corresponds to `UTOOL[1]`.

> ***NOTE:*** *One of the reasons for why I create tiny one-line programs for things like* `UFRAME`*s and* `UTOOL`*s is that the instructions themselves don't provide any context beyond the numeric identifier. For example, let's say I used* `UFRAME[5]` *for some fixture. I write this in my code:* `UFRAME_NUM=5`*. When I come back to this program a month or a year later, will I remember what*

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

`UFRAME[5]` *is for? Probably not, but I would know what a subroutine called* `UFRAME_PICK` *or* `UTOOL_GRIPPER` *does.*

Add some new lines and `CALL` the appropriate subroutines. Your program might look like this if you extracted the counter logic out:

```
CALL UTOOL_GRIPPER ;
CALL RESET_COUNTER ;
LBL[1] ;
CALL UFRAME_PICK ;
J P[1] 100% CNT0 ;
CALL GRIP ;
CALL UFRAME_PLACE ;
J P[2] 100% CNT0 ;
CALL UNGRIP ;
CALL INCREMENT_COUNTER ;
JMP LBL[1] ;
```

Run your program. You'll probably run into an error when the robot gets to the first motion statement. This is because your current user frame, `UFRAME[1]` , set by the `CALL UFRAME_PICK` statement, does not match the frame in which `P[1]` was recorded, `UFRAME[0]` or `WORLD`.

Positions are picky. Your programs must ensure that the correct `UFRAME` and `UTOOL` are active before issuing any motion instructions. If you had used a Position Register (`PR[]`), you wouldn't have gotten an error, but your robot may have moved to an unexpected location. It's important to keep these tools and frames straight.

Fix the mismatch by manually setting `UFRAME[0]` active. Press `SHIFT+COORD` and enter 0 when `Frame` is highlighted. Press `ENTER`. Now `STEP` through the motion statement to `P[1]` to bring your robot to that position. Set `UFRAME[1]` active and touch up `P[1]`. The editor will have you confirm you want to use frame 1 for this position (enter

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

1 then hit ENTER.) Do the same thing for P[2] (activate WORLD, step to P[2], activate UFRAME[2], then touch it up).

Normally you would setup your tool and user frames before teaching any points, but I didn't want to throw too many steps at you before we got the robot moving. Let's set them up now.

Creating a Tool Frame

Keep your editor open in the left pane and bring up the list of Tool Frames in the right window via SETUP > Frames. If you're not looking at list of Tool Frames, you may have to use F3 OTHER to get from the current type of Frames to Tool Frames.

They're all uncommented and zeroed out by default, but we're going to go in and modify UTOOL[1] and see how it affects our program.

Press F2 DETAIL on UTOOL[1] to edit this tool frame. Feel free to label it, but what we're really focused on here is the method for teaching it. The most common way to teach a Tool Frame is the Three Point method. This method has you record the same position in three different orientations. With this information the robot can determine where that point is relative to the faceplate.

Another common method is the Direct Entry method. Change the method to Direct Entry with F2 METHOD, and you'll see a list of components. If you've determined the components of your TCP from CAD, enter them here. If you don't actually have an EOAT, let's enter a value here to see the effect. Maybe enter something like 200.0 for the Z-component on a big robot; use something smaller (like 25.0 or 50.0) if you're on a small one.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

REMEMBER: *This value is the offset from the robot faceplate to some point in space (the Tool Center Point or TCP) that you want to drive to all of your positions that use this* `UTOOL`.

Run your program again. If you entered `200.0` you should see the robot move almost exactly as before. The faceplate is just 200mm further away from each point.

Creating a User Frame

Let's move on to our User Frames. If you're still editing a tool frame, press `PREV` to go back to the list. Open the list of User Frames by hitting `F3 OTHER` and choose User Frame.

Setting up User Frames is very similar to setting up Tool Frames. You hit `DETAIL`, give it a label, then choose a method for teaching. The most common method is again the Three Point method, but the procedure is different. You first record an Origin point (the front right corner of a fixture table, for example). Then you record a point in the +X direction (a little way down the right edge of the table). Finally, you record a point in the +Y direction (a bit to the left of your second point, touching the table surface). Based on these three points, the robot can determine the location and orientation of your frame.

NOTE: The first point determines the location (X, Y and Z) of your frame. Once the second point is added, the robot knows rotation around the Y- and Z- axes. The last point gives you rotation around the X-axis.

We're going to use the Direct Entry method to illustrate the relationship between frames and positions relative to them.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Go back to the editor and cursor to the position index for `P[1]` and then hit the `F4 Position` softkey to bring up this position's components. Go back to the frame editor and enter these values exactly as shown into `UFRAME[1]` with one difference: leave the W-value at 0 so that +Z aligns with the robot `WORLD` coordinate system. Hit `PREV` when you're done to get back to the list of frames and do the same thing for `P[2]` and `UFRAME[2]`.

> ***REMEMBER:*** *the points we recorded were relative to the* `WORLD` *user frame, which just so happens to be the same reference point for all user frames.*

We would have an issue if we simply ran our program now. Our points *were* relative to `WORLD`. They were relative to the same physical point in space even after we made them relative to the all-zero user frames 1 and 2. Where do you think they are now that our user frames have moved? How do you think you might change each position to bring the robot back to where it was?

A million points for you if you guessed that we should zero the positions out. The frame origins themselves are right where those points used to be. A point at `(0,0,0)` would be right at the frame origin.

Go back to the editor and zero out all components except for W. (**REMEMBER**: W indicates what direction our position's Z-axis will point. If the frame's Z-axis points up and our tool frame's Z-axis points down, W should be +/- 180.)

Run your program again. The robot should go through the same motions it did before we modified the user frames. If not, go back and check your work. The frame coordinates should be the old positions, and your positions should be 0 (except for W) to align with the frame origins.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Now that we've fully embraced tool frames and user frames, let's improve our robot's motion with reasonable approaches and retreats.

Motion Options: Offsets

Our jobs would be much easier if robots could always move directly from point A to point B. Unfortunately that's not generally the case. Depending on the application, they'll probably have to at least approach and retreat each fixture from some vector.

You could accomplish this by recording a few new points, but there's an easier way. We can copy and paste the motion statements we created earlier and simply extend them to include additional offsets for these approaches and retreats.

Frame Offsets

Frame Offsets essentially create a new destination by applying an offset to the programmed destination with respect to the current *User Frame*.

The offset value is stored in a *Position Register*. These are similar to the numeric registers we used earlier, but they store positions instead of numbers.

A Frame Offset is added with the `Offset,PR[]` motion option and makes a simple motion statement look like this:

```
J P[1] 100% CNT0 Offset,PR[1] ;
```

Basically this statement is telling the robot to go to the result of adding `PR[1]` to `P[1]` with respect to the current user frame. Let's do an example.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Offset Math

Imagine this is position 1…

```
X:  100.0      Y:  200.0      Z:  300.0
W:  180.0      P:    0.0      R:    0.0
```

…and this is `PR[1]`:

```
X:    0.0      Y:    0.0      Z:  100.0
W:    0.0      P:    0.0      R:    0.0
```

The actual destination where the robot TCP will end up is here:

```
X:  100.0      Y:  200.0      Z:  400.0
W:  180.0      P:    0.0      R:    0.0
```

Tool Offsets

Tool Offsets are similar, but, as you might suspect, the offset is taken with respect to the *Tool Frame* instead of the *User Frame*. These are added with the `Tool_Offset,PR[]` instruction and look like this:

```
J P[1] 100% CNT0 Tool_Offset,PR[1] ;
```

Where do you think the robot will end up if we use the same values for `P[1]` and `PR[1]`?

If you guessed the same position, you would unfortunately be wrong given our current tool frame. It's a little hard to understand at first, but the robot will actually end up here:

```
X:  100.0      Y:  200.0      Z:  200.0
```


Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

```
W:  180.0      P:    0.0      R:    0.0
```

The `180.0` yaw-value indicates that the `UTOOL` +Z direction is pointing in the exact opposite direction as the `UFRAME` +Z direction. Since the offset is applied with respect to the tool orientation, we are actually moving in the *negative* Z-direction with respect to the frame.

> ***NOTE:*** *It's conventional for tool frames to have +Z going in to the workpiece. The default* `UTOOL` *follows this convention with +Z going away from the robot faceplate. When following this convention, we will generally need to use negative Z-offsets for approaches and retreats.*

> ***TIP:*** *I generally prefer Tool Offsets to Frame Offsets when dealing with approaches and retreats (although Frame Offsets would work just fine in most cases). It seems to me that since the vector of approach and retreat are usually dependent on the tool itself and not the fixture, the offset should be relative to the tool's orientation as well, not the fixture's.*

Adding Offsets

Let's setup some Position Registers before adding the statements that will make up our approaches and retreats. Bring up the `DATA` screen and then use `F1 TYPE` to bring up the Position Register list. Label position registers 1 through 4 as follows:

1. Pick AP TO
2. Pick RT TO
3. Place AP TO

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

4. Place RT TO

> ***NOTE:*** *You could obviously name these however you want, but I usually follow this naming convention: AP for approach, RT for retreat and TO for Tool Offset (FO would indicate a Frame Offset).*

We now have to make sure these offsets have values. (You'll get an error when trying to execute a motion statement with uninitialized data.) Let's just use a `-100` Z-component and `0`s for all the rest like so:

```
X:    0.0      Y:    0.0      Z: -100.0
W:    0.0      P:    0.0      R:    0.0
```

Now that the position registers are setup, we have to add a few more motion statements: pick approach, pick retreat, place approach and place retreat.

`COPY/PASTE` is probably the best tool for this job. Move the cursor to the line that moves to `P[1]` and use `EDCMD > COPY/PASTE > SELECT > COPY` to copy that line. Immediately hit the `PASTE` softkey to paste the statement above the current line.

> ***NOTE:*** *You don't have to insert a new line. The editor adds new lines as necessary when pasting.*

Pasting a motion-statement requires one more step. The editor wants to know how much of the statement you want to copy over:

- `POSID` - everything, including the destination (e.g. `J P[1] 100% CNT0`)
- `LOGIC` - everything but the destination (e.g. `J P[...] 100% CNT0`)
- `POSITION` - everything, but create a new position for the destination (e.g. `J P[3] 100% CNT0`)

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Since we want to re-use `P[1]`, choose `POSID`. Paste with `POSID` again with the cursor on the `CALL UFRAME_PLACE` for pick retreat. Then paste `LOGIC` a couple of times where appropriate for the pick approach and pick retreat. You'll have to fill in the index of 2 for the destination on these two statements. Your program should look like this when you're done:

```
CALL UTOOL_GRIPPER ;
CALL RESET_COUNTER ;
LBL[1] ;
CALL UFRAME_PICK ;
J P[1] 100% CNT0 ;
J P[1] 100% CNT0 ;
CALL GRIP ;
J P[1] 100% CNT0 ;
CALL UFRAME_PLACE ;
J P[2] 100% CNT0 ;
J P[2] 100% CNT0 ;
CALL UNGRIP ;
J P[2] 100% CNT0 ;
CALL INCREMENT_COUNTER ;
JMP LBL[1] ;
```

This looks awfully redundant, but it will make more sense once we add offsets to most of these motion statements.

Cursor over the end of the first motion statement, press `F4 CHOICE` and find the `Tool_Offset,PR[]` motion option. Enter an index of `1` which corresponds to `PR[1:Pick AP TO]`. Use `PR[2:Pick RT TO]` on the motion statement after `CALL GRIP` and add the Tool Offsets for the place approach and retreat as well.

Here's the result (I added a blank line between pick and place for clarity):

```
CALL UTOOL_GRIPPER ;
CALL RESET_COUNTER ;
LBL[1] ;
```


Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

```
CALL UFRAME_PICK ;
J P[1] 100% CNT0 Tool_Offset,PR[1:Pick AP TO] ;
J P[1] 100% CNT0 ;
CALL GRIP ;
J P[1] 100% CNT0 Tool_Offset,PR[2:Pick RT TO] ;
;
CALL UFRAME_PLACE ;
J P[2] 100% CNT0 Tool_Offset,PR[3:Place AP TO] ;
J P[2] 100% CNT0 ;
CALL UNGRIP ;
J P[2] 100% CNT0 Tool_Offset,PR[4:Place RT TO] ;
CALL INCREMENT_COUNTER ;
JMP LBL[1] ;
```

If you run your program now, you should see the robot approach and retreat the pick and place points by `100mm`.

There's really no reason for the robot to slow down so much when approaching and retreating, so let's round the corners by changing the termination type on those motion statements to `CNT100`.

> ***TIP:*** *I try to use* `CNT100` *whenever possible to keep the robot as smooth and fast as possible. Use lower* `CNT`*-values if you have to avoid obstacles or need the robot to slow down at certain points.*

It may not be obvious, but the robot is actually taking a curved path for its approaches and retreats. This is because of the Joint motion type. Let's change a few of these to Linear motion statements so that the robot approaches and retreats in a straight line.

But first, a short interlude on the difference between Joint and Linear motion…

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Joint Motion

Take two points A and B. Each one has a unique set of joint angles. A Joint motion statement will simply move each joint from its starting position to its final position such that they all end up where they need to be at the same moment. The slowest joint (or the one that needs to move the most) will be the limiting factor.

> ***EXAMPLE:*** *Say we have a 2-axis robot where* `J1=0` *and* `J2=0` *for position A and* `J1=180` *and* `J2=90` *for position B. The joints will rotate simultaneously as the robot moves from A to B, but* `J1` *will move twice as fast. At the half-way point,* `J1` *will equal* `90` *and* `J2` *will equal* `45`.

The motion of the TCP may be anything but straight, but it's probably the most efficient.

Linear Motion

Linear motion calculates a path between points A and B such that the TCP moves in a straight line. Some axes may need to work quite a bit harder to maintain this straight-line path, but moving in straight lines is often necessary for getting in and out of tight spaces, avoiding obstacles, etc. You may find that linear motion is impossible between two positions (e.g. if there is a wrist configuration change), so you may have to resort to a Joint move (or the `WJNT` motion option – see the HandlingTool manual).

> *…and now back to our regularly scheduled program:*

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Highlight the `J` and then hit `F4` for `CHOICE`. Select `L` for Linear, and you should see the joint speed change to a scaled percentage of the robot's maximum linear speed. Feel free to change each speed to something that feels appropriate.

Your program should now look something like this:

```
CALL UTOOL_GRIPPER ;
CALL RESET_COUNTER ;
LBL[1] ;
CALL UFRAME_PICK ;
J P[1] 100% CNT100 Tool_Offset,PR[1:Pick AP TO] ;
L P[1] 250mm/sec CNT0 ;
CALL GRIP ;
L P[1] 250mm/sec CNT100 Tool_Offset,PR[2:Pick RT TO] ;
;
CALL UFRAME_PLACE ;
J P[2] 100% CNT100 Tool_Offset,PR[3:Place AP TO] ;
L P[2] 250mm/sec CNT0 ;
CALL UNGRIP ;
L P[2] 250mm/sec CNT100 Tool_Offset,PR[4:Place RT TO] ;
CALL INCREMENT_COUNTER ;
JMP LBL[1] ;
```

Notice that I have slowed down both the final moves before pick and place and their retreats to `250mm/sec`, but the robot still moves as fast as possible between the two stations.

If you run your program again you should see the robot move smoothly between pick and place.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Payloads

While not strictly necessary, it's a best-practice to always have the robot Payload set correctly.

What is a Payload? A Payload Schedule is a setting that tells your robot the physical properties (mass, center of gravity, inertia) of anything connected to the faceplate (e.g. the EOAT, any products that robot has gripped, etc.)

Why do we set Payloads? Without getting too deep into the math and science of making the robot move, your intuition may tell you that the robot has to work harder when carrying a 100kg weight vs. carrying nothing. If that's the case, it shouldn't surprise you that issues may result if you *tell* the robot it's carrying 100kg when it's actually carrying nothing (and vice-versa). You may even find that the robot's cycle time decreases dramatically when the activated payload is lowered.

Your payload settings also greatly affect the performance of Collision Guard. This feature basically works by comparing the actual torque on each axis vs. what it *expects* to measure based on the current payload. If the difference between those two measurements (the *disturbance torque*) is greater than a set threshold, the robot will post a collision detection. As you might expect, an inaccurate Payload will throw those calculations off, providing the opportunity for false-positives and late collision detection.

How do we set Payloads? Payload schedules are configured on the `Menu > System > Motion` screen. You can manually activate a payload from this screen, or, preferably, use `PAYLOAD[]` instructions in your programs to explicitly set the correct schedule when appropriate (e.g. on initialization and when the payload changes during a pick or place).

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

The PAYLOAD instruction

Add a new line immediately after the calls to `GRIP` and `UNGRIP`. These are the two points where "what's hanging off the end of the robot" changes.

Add a `PAYLOAD[]` instruction from the Payload instruction menu and enter an index of `2` after the call to `GRIP`. Enter a `PAYLOAD[1]` instruction after the call to `UNGRIP`.

> ***NOTE:*** *We could have chosen any of the ten available schedules for our empty and full gripper conditions, but I usually use* `PAYLOAD[1]` *for the empty gripper and the others for gripper plus product A, gripper plus product B, gripper plus product A and product B, etc.*

Let's configure our payload schedules now. Bring up the `PAYLOAD` configuration menu via `MENU > System > Motion`. By default they're all set to the maximum mass capacity of the robot, and all the center of gravity and inertia values are blank.

You can modify payload schedule `1` with `F2 DETAIL` and enter a value for the mass of the gripper. Hit `PREV` to get back out onto the main screen and then enter a value for the mass of the gripper and the product combined on `PAYLOAD[2]`.

> ***NOTE:*** *If you know the center of gravity and inertia values, great. If not, no big deal. I've found that mass is generally the most important value to have (and the easiest to get – you do have a scale on-site, don't you?)*

Once you're back on the `PAYLOAD` screen you can activate payload 1 by pressing the `SETIND` above `F5`. With the empty gripper payload activated, you can go back and view your program:

```
CALL UTOOL_GRIPPER ;
```


Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

```
CALL RESET_COUNTER ;
LBL[1] ;
CALL UFRAME_PICK ;
J P[1] 100% CNT100 Tool_Offset,PR[1:Pick AP TO] ;
L P[1] 250mm/sec CNT0 ;
CALL GRIP ;
PAYLOAD[2] ;
L P[1] 250mm/sec CNT100 Tool_Offset,PR[2:Pick RT TO] ;
;
CALL UFRAME_PLACE ;
J P[2] 100% CNT100 Tool_Offset,PR[3:Place AP TO] ;
L P[2] 250mm/sec CNT0 ;
CALL UNGRIP ;
PAYLOAD[1] ;
L P[2] 250mm/sec CNT100 Tool_Offset,PR[4:Place RT TO] ;
CALL INCREMENT_COUNTER ;
JMP LBL[1] ;
```

These instructions are similar to the `UFRAME_NUM=...` and `UTOOL_NUM=...` instructions we encountered earlier in that they unfortunately don't convey any meaning. Let's refactor our `PAYLOAD` statements into their own tiny programs called `PAYLOAD_EMPTY` and `PAYLOAD_FULL` and replace them in our main program to help our future selves and colleagues.

> ***NOTE***: *You may think I'm going a little overboard here with the tiny one-line programs, but trust me, it's worth your while. Take these payload programs for example. What if you needed to add a "dry cycle" capability to your robot, the ability for the robot to go through the motions without actually picking and dropping products. It's very easy to write and test a* `PAYLOAD_FULL` *program that functions appropriately with a dry cycle bit enabled or disabled, and you don't have to worry about muddying up the rest of your code with "dry cycle support logic" wherever the payload gets modified.*

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

We're at a point where our motion looks good, but I don't like how the robot is just blindly running between pick and place. What if the part's not there? What if the gripper fails to grip or there's already a part sitting at the place fixture?

Your robot is only as intelligent as you've programmed it to be. Let's take a look at conditionals and how they can be used to detect and handle errors gracefully.

Conditionals

We've made a bunch of assumptions in our little program. Here's a quick list (there may be more):

1. The part is present at the pick fixture
2. The robot gripper is empty and ready to grip
3. The robot gripper gripped the part successfully
4. The pick fixture is empty after the robot retreated from it
5. The place fixture is empty and ready for a part to be placed
6. The robot let go of the part correctly
7. The part made it onto the place fixture correctly

You may not necessarily need to check and handle *all* of these conditions, but it's probably a good idea.

Let's at least go through the exercise of making sure our pick fixture has a part before picking and making sure our place fixture is clear before placing. I'll leave the rest up to you.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

The IF-statement

We've already used a `JMP`-statement to *unconditionally* jump from one part of our program to another. We can also write an `IF`-statement that only jumps when a boolean expression evaluates to true.

Here's what an `IF`-statement looks like in TP: `IF <boolean expression>,<action>`. The boolean expression is probably some sort of comparison (e.g. `R[1]>5` or `DI[1]=ON`), and the action is a `JMP` or `CALL`.

> ***NOTE:*** *There are a couple of other options for actions if you're using mixed logic* `IF (...)`. *Feel free to explore the actions available to mixed-logic* `IF`*-statements on your own.*

Let's assume we have a couple of Digital Input signals that tell us if there is a part present at our pick and place fixtures: `DI[1:Part at Pick]` and `DI[2:Part at Place]`. (Feel free to use Flags or Robot Inputs if your Digital Inputs aren't mapped.) See if you can figure out how to label those inputs on the I/O status screen.

Add a new line before the move to pick approach and insert an `IF ...=...` statement from the `IF/SELECT` instruction menu. Choose `DI[1:Part at Pick]` for the left side of the comparison and `OFF` for the right side of the comparison. Make the action jump to a yet-to-be-created `LBL[501]`.

> ***NOTE:*** *You could have labeled your error branch with any index, but I like to keep my error* `LBL`*s in the* `500`*-range in the spirit of [HTTP status codes](). I like to use the convention of having my error branches correspond to top-level labels on a one-to-one basis: e.g.* `LBL[501]` *for an error branch that will jump back to* `LBL[1]`*,* `LBL[502]` *for* `LBL[2]`*, etc.*

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Add a few lines to the bottom of your program before adding a `LBL` and `JMP` instruction like this:

```
LBL[501] ;
;
JMP LBL[1] ;
```

What should we do when we get to this error branch? Lots of things I'm sure, but let's post a simple alarm message that forces the operator to reset (and hopefully fix) the error and restart before having the robot retry the condition that caused the fault in the first place.

User Alarms

FANUC provides a simple feature for displaying a short alarm message and pausing the robot with one instruction: User Alarms.

Press `MENU > Setup` and then press `F1` for `TYPE` and choose User Alarm. You'll see a list of 10 different alarms. Hit `ENTER` on the message area for alarm 1 and write a short error message, something like "Part missing at pick." While we're here, let's setup another user alarm for the case where the place fixture is not clear in `UALM[2]`, "Place fixture not clear."

Add a `UALM[]` instruction (from the Miscellaneous list) on the line after `LBL[501]` with an index of `1`. Your program should look like this:

```
CALL UTOOL_GRIPPER ;
CALL RESET_COUNTER ;
LBL[1] ;
IF DI[1:Part at Pick]=OFF,JMP LBL[501] ;
CALL UFRAME_PICK ;
```


Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

```
J P[1] 100% CNT100 Tool_Offset,PR[1:Pick AP TO] ;
L P[1] 250mm/sec CNT0 ;
CALL GRIP ;
CALL PAYLOAD_FULL ;
L P[1] 250mm/sec CNT100 Tool_Offset,PR[2:Pick RT TO] ;
;
CALL UFRAME_PLACE ;
J P[2] 100% CNT100 Tool_Offset,PR[3:Place AP TO] ;
L P[2] 250mm/sec CNT0 ;
CALL UNGRIP ;
CALL PAYLOAD_EMPTY ;
L P[2] 250mm/sec CNT100 Tool_Offset,PR[4:Place RT TO] ;
CALL INCREMENT_COUNTER ;
JMP LBL[1] ;
;
LBL[501] ;
UALM[1] ;
JMP LBL[1] ;
```

Run your program. You should see the task jump down to our error branch at `LBL[501]` when the `IF`-statement is evaluated. Execution pauses at our `UALM[1]` statement, and you should see the error message you configured at the top of the screen.

Put your robot into `STEP` mode, reset your faults and press `SHIFT+FWD` again. You should see it move down to the `JMP`-statement before coming back up to the top of the program at `LBL[1]`. If you execute the `IF`-statement again, it'll just keep coming right back down to `LBL[501]` until that digital input turns `ON`.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Simulating Inputs

Since you probably don't actually have a sensor wired in to DI[1], we can force the input into its ON-state to satisfy our program. Go the Digital Input status screen (I/O > Type > Digital) and look at the SIM column. Cursor over so that you're on the U in-line with digital input 1 and you should see the Simulate/Unsim softkeys above F4 and F5. Press F4 to simulate. Notice the U changes to an S, indicating that this input is now simulated. You can now cursor over to the STATUS column and use the F4 and F5 ON/OFF softkeys to toggle the input, overriding its actual state.

> ***TIP:*** *Be careful, especially when running in AUTO. You could easily crash the robot if you simulate the wrong input and forget to unsimulate it. There's an easy way to unsimulate all I/O:* FCTN > UNSIM ALL I/O.

Go ahead and turn DI[1] ON and run your program again. You should see it working like it did before, simply blowing right by the IF-statement because the error condition was not satisfied.

Add another LBL, an IF-statement and an error branch to make sure the place fixture is clear before placing like so:

```
CALL UTOOL_GRIPPER ;
CALL RESET_COUNTER ;
LBL[1] ;
IF DI[1:Part at Pick]=OFF,JMP LBL[501] ;
CALL UFRAME_PICK ;
J P[1] 100% CNT100 Tool_Offset,PR[1:Pick AP TO] ;
L P[1] 250mm/sec CNT0 ;
CALL GRIP ;
CALL PAYLOAD_FULL ;
L P[1] 250mm/sec CNT100 Tool_Offset,PR[2:Pick RT TO] ;
 ;
```


Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

```
LBL[2] ;
IF DI[2:Part at Place]=ON,JMP LBL[502] ;
CALL UFRAME_PLACE ;
J P[2] 100% CNT100 Tool_Offset,PR[3:Place AP TO] ;
L P[2] 250mm/sec CNT0 ;
CALL UNGRIP ;
CALL PAYLOAD_EMPTY ;
L P[2] 250mm/sec CNT100 Tool_Offset,PR[4:Place RT TO] ;
CALL INCREMENT_COUNTER ;
JMP LBL[1] ;
;
LBL[501] ;
UALM[1] ;
JMP LBL[1] ;
;
LBL[502] ;
UALM[2] ;
JMP LBL[2] ;
```

> ***NOTE:*** *The* `LBL[2]` *just before the check allows us to jump right back to this point. We wouldn't want to jump back to* `LBL[1]` *and try and pick again.*

Run your program. Test the failing condition by simulating `DI[2]=ON`, and then unsimulate it once you're satisfied the logic is working correctly.

Remarks and Comments

The `UALM` instruction is another one of those instructions that fails to convey meaning. Where I would usually extract this sort of thing into a well-named subroutine, let's use a remark instead.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Insert a new line before each `UALM` and then insert a Remark instruction via the Miscellaneous instruction menu. You'll see a new instruction that begins with the `!` character. You can write whatever you want after this character; it will be ignored by the interpreter. These are merely comments for your reference. Here's how my error branches look now:

```
LBL[501] ;
! Part not present at pick ;
UALM[1] ;
JMP LBL[1] ;
;
LBL[502] ;
! Place fixture not clear ;
UALM[2] ;
JMP LBL[2] ;
```

WAIT-statements

Wouldn't it be better if we could prevent the error from occurring in the first place? What if the robot got to that point in the program just a second or two before the fixture was ready? Let's add some flexibility and allow the robot to wait for a bit before jumping down to those error branches.

Cursor up to the first `IF`-statement and replace it with a `WAIT ...=...` statement. We have to change the condition to `DI[1:Part at Pick]=ON` since `WAIT` statements only proceed to the next line when the condition is `TRUE`.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

NOTE: *There are two types of* `WAIT`*-statements: those that wait for a condition (like the one we just added), and those that wait for a specified duration:* `WAIT ...(sec) ;`

WAIT-statements with a TIMEOUT

Your robot will wait here *forever* until the `WAIT`-statement's condition becomes true. Sometimes this works well, but oftentimes it's better to post an error after the robot has been waiting for too long.

Cursor over to the end of the statement. You should see the `F4 CHOICE` softkey. Add the `Timeout-LBL[]` option and have it jump to the `LBL[501]` error branch.

The default `TIMEOUT` duration is 30 seconds, but you can change this on the `MENU > System > Config` screen.

NOTE: *You can also change the* `$WAITTMOUT` *system variable directly from the* `MENU > System > Variables` *screen (units are in tenths of seconds) or with an assignment statement.* ***BE CAREFUL!*** *Most system variables should be left alone.*

Once you change both `IF`-statements to `WAIT`s with timeouts (remembering to flip the boolean expression logic), your program should look like this:

```
CALL UTOOL_GRIPPER ;
CALL RESET_COUNTER ;
LBL[1] ;
WAIT DI[1:Part at Pick]=ON TIMEOUT,LBL[501] ;
CALL UFRAME_PICK ;
J P[1] 100% CNT100 Tool_Offset,PR[1:Pick AP TO] ;
L P[1] 250mm/sec CNT0 ;
```


Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

```
CALL GRIP ;
CALL PAYLOAD_FULL ;
L P[1] 250mm/sec CNT100 Tool_Offset,PR[2:Pick RT TO] ;
;
LBL[2] ;
WAIT DI[2:Part at Place]=OFF TIMEOUT,LBL[502] ;
CALL UFRAME_PLACE ;
J P[2] 100% CNT100 Tool_Offset,PR[3:Place AP TO] ;
L P[2] 250mm/sec CNT0 ;
CALL UNGRIP ;
CALL PAYLOAD_EMPTY ;
L P[2] 250mm/sec CNT100 Tool_Offset,PR[4:Place RT TO] ;
CALL INCREMENT_COUNTER ;
JMP LBL[1] ;
 ;
LBL[501] ;
! Part not present at pick ;
UALM[1] ;
JMP LBL[1] ;
 ;
LBL[502] ;
! Place fixture not clear ;
UALM[2] ;
JMP LBL[2] ;
```

Our little program is more intelligent, but it's also quite long and somewhat complex. We now have four labels, four places where we can jump to other parts of the code, and the robot is doing two very different things.

It's be manageable now, but it's definitely harder to fully grasp the function of the program at a glance. You can imagine how things would quickly get out of hand if we started handling all the gripper checks, back-checking part-presence, etc.

Let's refactor our `WAIT` and error logic into a couple of new programs: `WAIT-_PICK_PRESENT` and `WAIT_PLACE_CLEAR`.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

`CUT` and `PASTE` the relevant lines from your main routine into the appropriate new routines. The only thing we need to add to them is a statement that will end the routine as soon as the condition is satisfied.

The END-statement

Add an `END` instruction immediately after each `WAIT`-statement so that the current program drops back into the calling program once the condition is satisfied.

A common anti-pattern is a `JMP` to some label that happens to be the last line of the program e.g. `JMP LBL[999]`. If you want to end the program, just use `END`.

Your `WAIT_PICK_PRESENT` routine should look like this:

```
LBL[1] ;
WAIT DI[1:Part at pick]=ON TIMEOUT,LBL[501] ;
END ;
 ;
LBL[501] ;
! Part not present at pick ;
UALM[1] ;
JMP LBL[1] ;
```

And your `WAIT_PLACE_CLEAR` routine should look like this:

```
LBL[2] ;
WAIT DI[2:Part at place]=OFF TIMEOUT,LBL[502] ;
END ;
 ;
LBL[502] ;
! Place fixture not clear ;
UALM[2]
JMP LBL[2] ;
```


Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

> ***NOTE:*** *label identifiers only have to be unique within the current program, so you could easily re-index the labels in* `WAIT_PLACE_CLEAR` *to* `1` *and* `501` *from* `2` *and* `502` *if you wanted to.*

Just like that our main program is significantly shorter, easier to read, and it's back to a single label:

```
CALL UTOOL_GRIPPER ;
CALL RESET_COUNTER ;
LBL[1] ;
CALL WAIT_PICK_PRESENT ;
CALL UFRAME_PICK ;
J P[1] 100% CNT100 Tool_Offset,PR[1:Pick AP TO] ;
L P[1] 250mm/sec CNT0 ;
CALL GRIP ;
CALL PAYLOAD_FULL ;
L P[1] 250mm/sec CNT100 Tool_Offset,PR[2:Pick RT TO] ;
;
CALL WAIT_PLACE_CLEAR ;
CALL UFRAME_PLACE ;
J P[2] 100% CNT100 Tool_Offset,PR[3:Place AP TO] ;
L P[2] 250mm/sec CNT0 ;
CALL UNGRIP ;
CALL PAYLOAD_EMPTY ;
L P[2] 250mm/sec CNT100 Tool_Offset,PR[4:Place RT TO] ;
CALL INCREMENT_COUNTER ;
JMP LBL[1] ;
```

I'll leave it as an exercise for the reader to add in the appropriate subroutines for gripper status and fixture back-checking. Your main routine may end up looking something like this:

```
CALL UTOOL_GRIPPER ;
CALL RESET_COUNTER ;
LBL[1] ;
```


Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

```
CALL WAIT_UNGRIP ;
CALL WAIT_PICK_PRESENT ;
CALL UFRAME_PICK ;
J P[1] 100% CNT100 Tool_Offset,PR[1:Pick AP TO] ;
L P[1] 250mm/sec CNT0 ;
CALL GRIP ;
CALL WAIT_GRIP ;
CALL PAYLOAD_FULL ;
L P[1] 250mm/sec CNT100 Tool_Offset,PR[2:Pick RT TO] ;
CALL WAIT_PICK_CLEAR ;
;
CALL WAIT_PLACE_CLEAR ;
CALL UFRAME_PLACE ;
J P[2] 100% CNT100 Tool_Offset,PR[3:Place AP TO] ;
L P[2] 250mm/sec CNT0 ;
CALL UNGRIP ;
CALL WAIT_UNGRIP ;
CALL PAYLOAD_EMPTY ;
L P[2] 250mm/sec CNT100 Tool_Offset,PR[4:Place RT TO] ;
CALL WAIT_PLACE_PRESENT ;
CALL INCREMENT_COUNTER ;
JMP LBL[1] ;
```

I'd probably recommend that you also refactor your pick and place logic into their own routines such that your main routine is actually quite short:

```
CALL RESET_COUNTER ;
LBL[1] ;
CALL PICK ;
CALL PLACE ;
CALL INCREMENT_COUNTER ;
JMP LBL[1] ;
```

This is where programming becomes less of a science and more of an art. The idea is to refactor as necessary such that your code is clean, easy to understand and easy to maintain.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

You have a pretty robust pick and place cycle here, but there are still a couple of important features that are crucial to a intelligent robot cell:

1. Homing the robot automatically upon restart
2. Cycle-stop functionality

Getting the Robot Home Safely

Most applications have a “home” or “perch” position for the robot. Your main routine usually expects the robot to be here before taking off, and it will probably go here before stopping its cycle during normal operation.

Some systems refuse to start the robot unless it’s already at home. This is arguably the safest option, but it’s probably painful from an operator’s perspective.

Personally I would rather give the robot the intelligence to bring itself home vs. relying on an operator who probably only jogs a robot once or twice a month. We can at least do our best to try and get home automatically from most of the robot’s usual hangouts, only falling back to operator intervention if absolutely necessary.

I save homing for last because it’s usually a pain in the ass. Depending on the complexity of your cell, it may take some serious thinking and programming. I wouldn’t be surprised to see more “homing” code than actual application code.

As with any programming task, there are many ways to do this: dropping “bread crumbs” throughout your code, recording and rewinding robot paths, looking at the robot’s current position, etc. Each method has pros and cons, but I almost always rely on the robot’s current position and attempt to retreat from known areas.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Getting the Robot's Current Position with `LPOS`

Let's create a new program called "AUTOHOME." This program will be responsible for getting the robot home safely from anywhere within it's normal cycle. The first step is to determine where the robot is.

You can get the robot's current position (with respect to the current User Frame and Tool Frame) by assigning `LPOS` (for linear position) to a position register. Something like this: `PR[5:LPOS]=LPOS ;`. We can then look at the various components (X, Y, Z, etc.) of that position register to draw conclusions about where the robot is in space.

> ***NOTE:*** *There is a* `JPOS` *value as well which returns the robot's current Joint-values.*

Let's first try to see if the robot's at the pick fixture. Add these lines to your `AUTOHOME` program:

```
CALL UFRAME_PICK ;
CALL UTOOL_GRIPPER ;
PR[5:LPOS]=LPOS ;
```

> ***NOTE:*** *Make sure to label* `PR[5]` *on the list of Position Registers(*`DATA > F1 (TYPE) > Position Registers`*).*

Run your program and then bring up the value of `PR[5:LPOS]` on the Position Registers `DATA` screen. (**REMEMBER:** `DATA > F1 (TYPE) > Position Registers`)

If your robot is currently at the pick position, the location should be all zeroes. If it was at the pick approach, X and Y would be roughly zero, and Z would be a bit higher. Remember how the pick position is all zeroes with respect to our pick User Frame?

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

With this in mind, we can add some `IF`-statements to quickly filter out conditions where the robot is definitely *NOT* at the pick fixture. If the robot makes it through all of them, it must be somewhere close by. These sorts of `IF`-statements are called [Guards]().

The tolerance for each component will be up to you, but let's add some guards that, once passed, will confirm that the robot is at the pick fixture. Start with the X-component.

Mixed Logic Instructions/Expressions

Add an `IF (...)` statement from the IF/SELECT instruction menu. FANUC refers to this type of expression (what's between the parentheses) as Mixed Logic. They are used to compose boolean expressions with a bit more flexibility than the simple comparisons we've been using so far.

Once you add the `IF (...)` statement, your cursor will be inside the parentheses, waiting for you to add items and operators to create a valid boolean expression. `INSERT` a `PR[i,j]` item using `5` for `i` and `1` for `j` so that it looks like this: `PR[5,1]`. This represents the first component (the X-component) of `PR[5]`. (You would use `PR[5,2]` for Y, `PR[5,3]` for Z, `PR[5,4]` for W, and so on.)

Next `INSERT` a < operator before adding a Constant of `-100`. Make the statement jump to the non-existent `LBL[500]` for now. It should look like this when you're done:

```
IF (PR[5,1]<(-100)),JMP LBL[500] ;
```

Let's read the statement: if the X-component of `PR[5:LPOS]` is less than negative one-hundred, jump to `LBL[500]`. Or, in other words, if the robot's not within roughly four inches of the pick user frame's origin, we know we're not at the pick fixture.

Add another instruction (by hand or with the `Cut/Copy EDCMD`) to validate the positive side of the X-component, giving the robot +/- `100mm` in the X-direction.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

We're going to validate the Y- and Z-components in this same fashion, but maybe with slightly different tolerances such that the end-result is something like this:

```
CALL UFRAME_PICK ;
CALL UTOOL_GRIPPER ;
PR[5:LPOS]=LPOS ;
IF (PR[5,1]<(-100)),JMP LBL[500] ;
IF (PR[5,1]>100),JMP LBL[500] ;
IF (PR[5,2]<(-50)),JMP LBL[500] ;
IF (PR[5,2]>50),JMP LBL[500] ;
IF (PR[5,3]<0),JMP LBL[500] ;
IF (PR[5,3]>200),JMP LBL[500] ;
```

If you're thinking "that's a lot of `IF`-statements," you're right, and I would agree with you. You could have created a massive mixed-logic statement like this: `IF (PR[5,1]<(-100) OR PR[5,1]>100 OR PR[5,2]<(-50) OR PR[5,2]>50 OR PR[5,3]<0 OR PR[5,3]>200),JMP LBL[500] ;`. However, I think it's 1) easier to read and write six short `IF`-statements, and 2) six separate statements are easier to debug with `STEP`-mode. (e.g. which condition is not satisfied?)

If the task makes it through all of those guards we can be confident that we are reasonably close to the pick position. Now we have to write code to retreat safely from there.

IF-THEN Blocks

Up until this point we have used simple `IF`-statements that can only have one action, usually a `JMP` or a `CALL`. These were our only options for a long time, but thankfully FANUC gave us `IF-THEN` blocks in v8.30 (I think?).

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

These sorts of statements work like the `IF`-style control structures you're probably used to if you're familiar with any other modern programming language. They look like this:

```
IF (<boolean expression>) THEN ;
<true branch>
ENDIF ;
```

They may also have an optional `ELSE`-branch:

```
IF (<boolean expression>) THEN ;
<true branch>
ELSE ;
<false branch>
ENDIF ;
```

This is a much more convenient way of writing:

```
IF (<boolean expression>),JMP LBL[<true label>] ;
<false branch>
JMP LBL[<end label>] ;
LBL[<true label>] ;
<true branch>
LBL[<end label>] ;
```

As I said before, the fewer labels and jumps the better. Let's let the controller take care of that stuff for us since it can.

Safe Retreat Path

Now that we've verified our robot is near the pick fixture, how do we want it to retreat? This will vary depending on the application, but let's do something simple here. Let's have the robot move straight up to a height of 200mm above the fixture (if necessary),

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

center itself (again, if necessary), and then move home directly. We'll do this with a few `IF-THEN` blocks.

Add an `IF (...) THEN` instruction from the IF/SELECT menu. Our expression will be `PR[5,3]<200`. We want the "true branch" to execute if the robot's TCP is lower than 200mm from the top of the table.

Add an `ENDIF` instruction (from the IF/SELECT menu) immediately after the `IF (...) THEN` and then insert two new lines above it. Remember how we saved the robot's current position to `PR[5:LPOS]` earlier? It's still there, and we can actually modify the Z-component of this position directly and have the robot move straight up from where it currently is.

Add an assignment instruction `...=...` from the Registers menu on the first blank line after your `IF`-statement. Use `PR[i,j]` for the receiver and set it to `PR[5,3]` then enter a Constant of `200` as the value. This will set `PR[5:LPOS]`'s Z-component to `200`. At this instant, the value of `PR[5:LPOS]` will be a point directly above the robot's current position with the same orientation.

Add a motion statement on the next line (`SHIFT+POINT`). Change the motion type to Linear (select `J` and change it to `L`) and change the destination to `PR[5]`. Set the speed to something cautious (like 100mm/sec) and use `CNT0` for the termination type. This motion statement will move the robot to a point that's exactly 200mm above the table.

Here's what your program should look like now:

```
CALL UFRAME_PICK ;
CALL UTOOL_GRIPPER ;
PR[5:LPOS]=LPOS ;
IF (PR[5,1]<(-100)),JMP LBL[500] ;
IF (PR[5,1]>100),JMP LBL[500] ;
IF (PR[5,2]<(-50)),JMP LBL[500] ;
```


Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

```
IF (PR[5,2]>50),JMP LBL[500] ;
IF (PR[5,3]<0),JMP LBL[500] ;
IF (PR[5,3]>200),JMP LBL[500] ;
IF (PR[5,3]<200) THEN ;
PR[5,3:LPOS]=200 ;
L PR[5:LPOS] 100mm/sec CNT0 ;
ENDIF ;
```

Jog the robot somewhere away from the pick fixture (outside the range of your guards) and then STEP through your program. You should see one of the guards throw an error complaining that `LBL[500]` doesn't exist. Next run through your main routine to the pick position and run `AUTOHOME` again. You should see the robot move up 200mm.

Similarly, let's add a couple of `IF-THEN` statements (and a couple of remarks) to center the robot in the X- and Y-components too:

```
CALL UFRAME_PICK ;
CALL UTOOL_GRIPPER ;
PR[5:LPOS]=LPOS ;
IF (PR[5,1]<(-100)),JMP LBL[500] ;
IF (PR[5,1]>100),JMP LBL[500] ;
IF (PR[5,2]<(-50)),JMP LBL[500] ;
IF (PR[5,2]>50),JMP LBL[500] ;
IF (PR[5,3]<0),JMP LBL[500] ;
IF (PR[5,3]>200),JMP LBL[500] ;
! we are at the pick fixture ;
! move up ;
IF (PR[5,3]<200) THEN ;
PR[5,3:LPOS]=200 ;
L PR[5:LPOS] 100mm/sec CNT0 ;
ENDIF ;
! move left/right ;
IF (PR[5,2]<>0) THEN ;
PR[5,2:LPOS]=0 ;
L PR[5:LPOS] 100mm/sec CNT0 ;
ENDIF ;
```


Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

```
! move forward/back ;
IF (PR[5,1]<>0) THEN ;
PR[5,1:LPOS]=0 ;
L PR[5:LPOS] 100mm/sec CNT0 ;
ENDIF ;
```

If the robot makes it through that whole program without faulting, you will know that the robot's final position is `(0, 0, 200)`. From here we should be able to get home with a simple Joint move. Let's save ourself some time later and create a subroutine for moving straight home named "HOME."

Moving Straight Home

We made our pick position relative to the pick user frame. We made the place position relative to the place user frame. Where should our home position be relative to?

I suppose it may depend on your application, but I would think that Home should generally be relative to the robot itself, the `WORLD` frame. You probably don't want the Home position to move if any frames were touched up.

Create a `UFRAME_WORLD` subroutine with this statement: `UFRAME_NUM=0 ;`. Let's use `PR[6]` for the Home position. Go ahead and label this on the Position Registers list.

We'll just use a simple and slow Joint move in this routine (with the correct `UTOOL`, of course.) Make sure your motion statement's destination is `PR[6:Home]` and then touch it up (probably somewhere safe between the two fixtures). Double-check to make sure you have the correct User Frame and Tool frame active and set an appropriate speed. Your `HOME` program should look like this:

```
CALL UFRAME_WORLD ;
CALL UTOOL_GRIPPER ;
```


Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

```
J PR[6:Home] 10% CNT0 ;
```

You can use this routine to quickly move Home from anywhere (no guarantees that it's safe to do so), and you'll be able to use `AUTOHOME` to try and retreat safely first.

Let's finish the retreat from the pick fixture in `AUTOHOME` by calling `HOME` before ending the program:

```
CALL UFRAME_PICK ;
CALL UTOOL_GRIPPER ;
PR[5:LPOS]=LPOS ;
IF (PR[5,1]<(-100)),JMP LBL[500] ;
IF (PR[5,1]>100),JMP LBL[500] ;
IF (PR[5,2]<(-50)),JMP LBL[500] ;
IF (PR[5,2]>50),JMP LBL[500] ;
IF (PR[5,3]<0),JMP LBL[500] ;
IF (PR[5,3]>200),JMP LBL[500] ;
! we are at the pick fixture ;
! move up ;
IF (PR[5,3]<200) THEN ;
PR[5,3:LPOS]=200 ;
L PR[5:LPOS] 100mm/sec CNT0 ;
ENDIF ;
! move left/right ;
IF (PR[5,2]<>0) THEN ;
PR[5,2:LPOS]=0 ;
L PR[5:LPOS] 100mm/sec CNT0 ;
ENDIF ;
! move forward/back ;
IF (PR[5,1]<>0 THEN ;
PR[5,1:LPOS]=0 ;
L PR[5:LPOS] 100mm/sec CNT0 ;
ENDIF ;
! robot at (0, 0, 200) ;
CALL HOME ;
END ;
```


Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

While our robot can now get home safely from the pick fixture, the non-existent `LBL[500]` is an obvious glaring error. The routine's already pretty long too, so let's refactor some of this pick fixture stuff before adding more functionality to it.

Returning Values from Subroutines

Most programming languages allow you to return values from functions: `two = add(1,1) ;`. Unfortunately there's no `RETURN`-statement in TP. The best we can do is designate a Numeric Register (or Flag, or `DO[]`, whatever) for this sort of thing and hope that nothing else is writing to that piece of data.

Let's designate `R[10]` our Status register, `R[10:Status]`. A common convention is to use `0` as the "normal" or "expected" status. Anything non-zero would indicate an unexpected or error condition.

Let's pull out the section of code that determines whether or not the robot is at the pick fixture into a new routine: `IS_AT_PICK`. We'll return `R[10:Status]=0` if the robot is near the pick fixture, and we'll return 1 otherwise.

`Cut/Copy` lines 1-9 from `AUTOHOME` into `IS_AT_PICK`, everything from `CALL UFRAME_PICK` to the last `IF (...)` statement. Add a `...=...` assignment as the last line, setting `R[10:Status]` to 0 before adding an `END`-statement. Lastly, add `LBL[500]` which simply sets `R[10:Status]` to 1.

```
CALL UFRAME_PICK ;
CALL UTOOL_GRIPPER ;
PR[5:LPOS]=LPOS ;
IF (PR[5,1]<(-100)),JMP LBL[500] ;
IF (PR[5,1]>100),JMP LBL[500] ;
IF (PR[5,2]<(-50)),JMP LBL[500] ;
IF (PR[5,2]>50),JMP LBL[500] ;
IF (PR[5,3]<0),JMP LBL[500] ;
```


Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

```
IF (PR[5,3]>200),JMP LBL[500] ;
R[10:Status]=0 ;
END ;
;
LBL[500] ;
R[10:Status]=1 ;
```

Jog the robot to the pick position and look at your list of Numeric Registers while running `IS_AT_PICK`. You should see `R[10:Status]` equal `0`. Set it to something else and run the program. It should return to `0`. Now jog the robot to somewhere outside of the guards' boundaries. You should see `R[10:Status]` equal `1`. This should prove to you that the routine is working.

> ***NOTE:*** *If you wanted to be really thorough, you could create your own* `TEST` *routines to do what I just described automatically. Program the robot to go to the pick position, run* `IS_AT_PICK`*, and then evaluate an* `IF`*-statement to ensure the value is* `0`*. Move the robot somewhere else and verify the return-value is non-zero. While you're at it, you may even refactor the* `LPOS` *logic out of the* `IS_AT_PICK` *routine and pass a Position Register ID to the routine (e.g.* `CALL IS_AT_PICK(5)`*). This would allow you to test to see if arbitrary positions are within the pick zone, not necessarily the robot's current position. But I digress… Unit testing is outside the scope of this book, but I highly recommend looking into it. I've written a few blog posts about the subject.*

We should now replace all that code that we extracted from our `AUTOHOME` program with a `CALL` and an `IF`-statement:

```
CALL IS_AT_PICK ;
IF R[10:Status]<>0,JMP LBL[500] ;
! we are at the pick fixture ;
! move up ;
```


Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

```
IF (PR[5,3]<200) THEN ;
PR[5,3:LPOS]=200 ;
L PR[5:LPOS] 100mm/sec CNT0 ;
ENDIF ;
! move left/right ;
IF (PR[5,2]<>0) THEN ;
PR[5,2:LPOS]=0 ;
L PR[5:LPOS] 100mm/sec CNT0 ;
ENDIF ;
! move forward/back ;
IF (PR[5,1]<>0 THEN ;
PR[5,1:LPOS]=0 ;
L PR[5:LPOS] 100mm/sec CNT0 ;
ENDIF ;
! robot at (0, 0, 200) ;
CALL HOME ;
END ;
```

While we're at it, let's pull all the retreat logic out into a new program, `RETREAT_FROM_PICK`, making sure to add calls to `UFRAME_PICK` and `UTOOL_GRIPPER`. Let's also change the `JMP LBL[500]` to `JMP LBL[2]` since it's not really an error if the robot is not at home. Our `AUTOHOME` program now looks like this:

```
CALL IS_AT_PICK ;
IF R[10:Status]<>0,JMP LBL[2] ;
CALL RETREAT_FROM_PICK ;
CALL HOME ;
END ;
```

Much cleaner! It will be much easier to add logic for the place fixture with these pieces in place.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Copying PROGRAMs

The logic for a program that tells us if we're at the place fixture will be very similar to IS_AT_PICK. The only difference will be the active User Frame and perhaps the guard limits. Let's create the IS_AT_PLACE program by creating a copy of IS_AT_PICK before modifying it.

Bring up the list of programs with the SELECT key. Cursor up or down until the IS_AT_PICK program is highlighted. Hit the F2 COPY softkey and name your copy IS_AT_PLACE before hitting ENTER and YES to the confirmation prompt. Your new program should now be listed. Highlight it and press ENTER to bring up the editor.

All we really need to do here is call the correct User Frame routine. Cursor over such that UFRAME_PICK is highlighted then use F4 CHOICE to change the program name to UFRAME_PLACE.

Test this routine in the same way you tested IS_AT_PICK. Jog the robot somewhere and run the routine. Check to see if the R[10:Status] value looks ok. Make sure to test case where R[10:Status] should return 1.

For brevity, let's make the retreat from the place fixture nearly the same as well. Create a copy of the RETREAT_FROM_PICK routine named RETREAT_FROM_PLACE. Replace the call to UFRAME_PICK with UFRAME_PLACE. We just need to add calls to these new programs in our main AUTOHOME program, and we should be pretty much good to go:

```
CALL IS_AT_PICK ;
IF R[10:Status]<>0,JMP LBL[2] ;
CALL RETREAT_FROM_PICK ;
CALL HOME ;
END ;
;
LBL[2] ;
```


Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

```
CALL IS_AT_PLACE ;
IF R[10:Status]<>0,JMP LBL[500] ;
CALL RETREAT_FROM_PLACE ;
CALL HOME ;
END ;
```

Jog the robot near the pick fixture and run `AUTOHOME`. You should see the robot retreat from the fixture (if necessary) and then move Home. Do the same thing near the place fixture. The robot should retreat correctly. If not, make sure you're calling the correct User Frame when appropriate.

The only remaining issue for us now is to figure out how to handle the case when the robot is not near either the pick or place fixture.

Not sure where we are

The way I see it, we have two options here:

1. **WARN** the user that the move home may be unsafe before moving directly home
2. **WARN** the user that we are at an unknown position and `ABORT`, requiring manual intervention.

For option 1 you'd simply post a `UALM` before calling `HOME`, but I don't really like this method. Can we really trust the operator to read the alarm and have their finger on the `HOLD` button?

Let's implement option 2 instead.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

UALMs, Revisited

We will want to tell the user why the robot's not starting up, so let's configure a User Alarm: `MENU > Setup > F1 (Type) > User Alarms` to do just that.

Setup a message for `UALM[3]`, something like "Manual homing required."

Now add the `LBL[500]` branch to your program with a simple `UALM[3]` statement. Your `AUTOHOME` program should now look like this:

```
CALL IS_AT_PICK ;
IF R[10:Status]<>0,JMP LBL[2] ;
CALL RETREAT_FROM_PICK ;
CALL HOME ;
END ;
;
LBL[2] ;
CALL IS_AT_PLACE ;
IF R[10:Status]<>0,JMP LBL[500] ;
CALL RETREAT_FROM_PLACE ;
CALL HOME ;
END ;
 ;
LBL[500] ;
! manual homing required ;
UALM[3] ;
```

The only issue I see here is that `AUTOHOME` will fall through to any calling programs after `UALM[3]` is acknowledged. Let's modify the behavior of these `UALM` such that it actually aborts the program for us.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Configuring `UALM`s

It is possible to configure the severity of User Alarms even though there is no easy-to-use interface for doing so. (Anyone at FANUC reading this?) We'll unfortunately have to set a system variable directly for this.

Open the System Variables screen via `MENU > System > Variables`. Scroll down to `$ualrm_sev` and press `ENTER`. `$ualrm_sev` is an array of `10` values, each corresponding to the associated `UALM[]`. You should see that they're all currently set to `6` which pauses the program and stops its motion. Change the `6` to a `3`. This will post the error and abort the program. Other values are described below:

$ualrmsev values

0 - No Action

2 - Pause program

3 - Abort program with error

4 - Stop program motion

6 - (Default) Pause program and stop its motion

8 - Cancel program motion

10 - Pause program and cancel its motion

11 - Abort program and cancel its motion

- *Add 16 to any value to turn off all servo motors*
- *Add 32 to apply the action to ALL programs and motions*
- *Add 64 for particularly nasty error conditions to require a cold start of the controller*

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

But what about the case when the robot is already home? We didn't check for this possibility. While we could easily write another `IS_AT_*`-type routine, let's use FANUC's built-in Reference Position feature to make it a bit easier on ourselves.

Reference Positions

The robot gives you 10 *Reference Positions*, basically positions you can setup such that the robot automatically turns `ON` an output when the robot is within some Joint-position tolerance and `OFF` otherwise. Let's set one up for our Home position.

First run your `HOME` routine to make sure the robot's at home. Then pull up the Reference Positions screen via `MENU > Setup > Reference Positions`. Next press `DETAIL` on the first reference position and press the `F5 RECORD` softkey with `SHIFT`. You'll see the robot's current joint positions populate each setting near the bottom. By default, the reference position has zero tolerance for being slightly off this position. You'll probably fall off of these exact joint angles as soon as you release the deadman switch. Let's fix that by giving a bit of tolerance to each joint position in the +/- column on the right… maybe 0.1 degree each.

Now we just need to assign an output to turn on when the robot is near Home. Cursor up and assign `DO[1]` to turn on for this reference position. You can bring up the Digital Output status screen (`MENU > I/O > Digital`) and verify that this output turns `ON` when you jog the robot to Home, and it turns `OFF` when you jog the robot away.

Lastly, let's add a simple `IF-THEN` block to the top of our `AUTOHOME` program to allow it to `END` if the robot is already Home:

```
IF (DO[1:At Home]=ON) THEN ;
```


Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

```
END ;
ENDIF ;
```

> ***NOTE:*** *You could have simply used* `DO[1:At  Home]`*as your condition, but we'll cover that a bit later.*

There you have it: a smart homing routine. Call this from the top of your main routine to ensure the robot is at Home before starting the pick and place cycle:

```
CALL AUTOHOME ;
CALL RESET_COUNTER ;
LBL[1] ;
CALL PICK ;
CALL PLACE ;
CALL INCREMENT_COUNTER ;
JMP LBL[1] ;
```

This application is *almost* done. The last thing we need to add is the capability to safely stop the cycle.

Cycle Stop and `ABORT`

We just went through the trouble of creating a routine that can safely home the robot from anywhere near its pick and place locations. Despite that effort, let's add some functionality to safely move the robot home upon some signal so that all that work probably won't get used much!

Let's start by creating the `CYCLE_STOP` routine. First, let's make an assumption that the robot will definitely be in a safe position to move directly home when this routine is called. If that's the case, here's all we need:

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

```
CALL HOME ;
ABORT ;
```

The ABORT-instruction is added from the Program Control instruction menu. You might guess that this statement aborts the current task's execution when this statement gets executed. It won't fall back to the calling program, so we should be able to just CALL CYCLE_STOP as necessary without worrying about any cleanup afterwards.

The question is, "Where do we add this?"

You might be tempted to add this check into the main routine: either just before the call to PICK or after the call to PLACE. And that would work just fine if the cycle stop signal is given mid-cycle. But what if the signal is given while the robot is waiting to pick? We probably don't want the robot to go through the effort of picking and placing again if it's been told to stop the cycle. If that's the case, we should only have to add a bit of code to WAIT_PICK_PRESENT.

The robot will currently hang on this program's WAIT-statement until one of two things are true: 1) the part-presence signal comes on, or 2) the WAIT-statement times out. Let's add a third possibility here where the WAIT-statement will fall through if our cycle stop signal comes on.

Mixed Logic WAIT-statement

Go to the SELECT menu and hit ENTER on WAIT_PICK_PRESENT to bring up an editor. Cursor up so that the cursor is on the same line as the WAIT-statement and replace the WAIT ...=... instruction with the WAIT (...) instruction. This statement allows us to create a more flexible boolean expression like our mixed logic IF (...) statements. We are going to make the expression evaluate to true when either DI[1:Part at Pick] or DI[3:Cycle Stop] turns ON.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Cursor in between the parentheses and `INSERT` a `DI[]`. Enter an index of `1`. Now you may be tempted to insert an = and then an `ON` value. You *could* do that, but you don't *have to*. Boolean I/O ports (like a `DI[]` or `RO[]`) themselves evaluate to `true` if they're `ON` when used in a mixed logic expression. There's no need to compare them to the `ON`-value. `IF (DI[1])` is the same as `IF (DI[1]=ON)`, so let's save ourselves a few keystrokes and leave it at that.

After you've added the `DI[1:Part at Pick]` value, cursor over to the right and add an `OR` operator. You can then add another `DI[]` for `DI[3:Cycle Stop]` to get this `WAIT`-statement:

```
WAIT (DI[1:Part at Pick] OR DI[3:Cycle Stop]) ;
```

This will fall through when either of those inputs goes high, so we'll just have to check if `DI[3:Cycle Stop]` is `ON` immediately afterwards, ending the program as we did before otherwise. (You'll have to re-add the `TIMEOUT`to `LBL[501]` just like you did before as well.)

Add a new line just before the `END`-statement. Add an `IF (...)` statement and make the condition `DI[3:Cycle Stop]`. Set the action to `CALL CYCLE_STOP`. Your program should now look like this:

```
LBL[1] ;
WAIT (DI[1:Part at pick] OR DI[3:Cycle Stop]) TIMEOUT,LBL[501] ;
IF (DI[3:Cycle Stop]),CALL CYCLE_STOP ;
END ;
;
LBL[501] ;
! Part not present at pick ;
UALM[1] ;
JMP LBL[1] ;
```

Remember that `CYCLE_STOP` will `ABORT` the task, so we don't have to worry about it falling through to that `END`-statement and the calling program, `PICK`. If it was

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

DI[1:Part at pick] which causes the WAIT to release, we'll simply END like we did before.

Naming, Revisited

One might argue that the name WAIT_PICK_PRESENT is no longer an accurate description of the functionality of this program. A better name might be WAIT_PICK_PRESENT_OR_CYCLE_STOP. I'll leave it as an exercise to the reader to rename this program and update the PICK program that calls it.

An Intelligent Robot

At this point you have what I would call a mature program and an intelligent robot. The main program ensures that the sequence does not start until the robot finds its way home. The pick and place cycles use the inputs available to alert the operator of any issues, and it watches for a stop signal at the most appropriate point, getting the robot home before aborting the task.

This should be the goal of every project: an intelligent robot that generally takes care of itself. I've seen many installations where the end-user is not happy with the automation, and it's almost always the result of a lazy (not necessarily bad) programmer.

Getting the robot to do the "normal" case is only half the job (or less). Handling all the extraneous "what if?" cases is what separates a job well done vs. one that gives robots and automation a bad name.

Let's do our industry a favor and handle all those edge-cases so our customers are happy.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

What You Have Accomplished

Let's sum up what you did:

- You explicitly set the `UFRAME` and `UTOOL` before doing any motion to prevent issues down the road.
- You went from having `WORLD` positions to having positions relative to custom, relevant and logical user frames.
- You created approach and retreat motions for pick and place with only two unique positions, favoring offsets over new points.
- You made sure to check part presence and gripper status when appropriate, handling errors before moving on.
- You used accurate Payloads to ensure optimal motion performance.
- You used `WAIT`-statements instead of `IF`-statements to provide your program with a little more flexibility.
- You learned to simulate inputs while testing.
- You progressively refactored bits of functionality into small, reusable, semantically-named subroutines so that your programs are easier to read and manage.
- You wrote code to automatically get the robot home from most positions within its normal cycle, only involving the operator when absolutely necessary.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

- You handled a normal request for a cycle stop, cleaning up by getting the robot home before releasing the task.

Believe it or not, these instructions and concepts probably constitute over 90% of the code I've written as a FANUC robot programmer:

- Subroutines and `CALL`-statements
- Label definitions
- Conditional and unconditional jumps
- Motion statements
- Payloads
- Frame and Tool Offsets
- `WAIT`-statements (with optional timeouts)
- User Alarms and their Configuration
- Error handling
- Homing with `LPOS` and direct `PR[i,j]` assignment
- Reference Positions
- Mixed Logic

You should now be well-equipped to tackle just about any material handling application.

I hope you enjoyed this crash course in TP programming. If you have any questions or comments, please feel free to email me. I'd love to hear your feedback.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Appendix A: Mapping I/O

Mapping I/O is easy, but it can be a little confusing if you've never done it before. Before we get into the details, let's start with a quick review of the types of I/O that are available to you on the robot.

- **Analog I/O** (`AI` and `AO`) signals are real numbers that represent voltages within the range of input or output module used (e.g. you could send a value of `5.5` to a `0` to `+10V` output card or read a value of `-3.1` from a `-10` to `0V` input card).
- **Digital I/O** (`DI` and `DO`) signals are boolean values (`ON` or `OFF`).
- **Group I/O** (`GI` and `GO`) signals allow you to interpret a group of input or output bits as an integer from its binary representation.
- **User Operator Panel** (UOP) signals (`UI` and `UO`) can control the robot or give the status of the robot operation.
- **Standard Operator Panel** (SOP) (`SI` and `SO`) signals are used internally by the controller's physical operator panel (you know, the big green button with the plastic covering on it).
- **Robot I/O** (`RI` and `RO`) signals are inputs and outputs delivered via the End Effector (EE) connector, useful for getting I/O to and from your End-of-Arm Tool (EOAT).
- **Flag** (`F`) signals are a useful bank of boolean values that don't go anywhere by default.

With these you can send and receive boolean signals, integer values or real values. What more could you ask for?

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Analog (real), Digital (boolean), Group (integer) and Flag (boolean) I/O are general-purpose that can be used for anything while UOP, SOP and Robot I/O (all boolean) signals have dedicated jobs on the controller.

For a typical project in 2018 you'll probably setup a connection between the robot and a PLC over Ethernet/IP (or maybe PROFINET or MODBUS, but the concepts are the same). You'll decide to share some number of input and output bits (e.g. 256), and then you'll map that connection to some Digital, Group and/or UOP signals. You might use Robot I/O to control your EOAT with the EE cable, and you might make use of some Flag bits internally.

Racks and Slots and Ports, Oh My!

Mapping I/O all comes down to understanding racks, slots and ports. Remember that these configuration screens were designed way back in the day when your I/O was actually wired up (with real wires!) to physical devices that were mounted on real physical racks inside the controller.

> ***NOTE:*** *Maybe you actually are using a physical input or output card, but in my experience it's much more common to handle everything over ethernet these days.*

Picture a cabinet with DIN rails running from top to bottom. The device mounted on the top rack farthest to the left would be at rack 1 slot 1. The next device would be rack 1 slot 2 and so on.

Each of these racks is for a specific type of I/O. A few examples:

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

- Rack 0 is for process I/O boards
- Rack 1 is for modular (Model A) I/O
- Rack 36 is for DCS
- Rack 89 is for Ethernet/IP

It would be nice if FANUC provided a drop-down menu on the configuration screen, but for now you'll have to manually enter these rack numbers.

Each entry in the I/O configuration table allows you to create a connection between some type of I/O (e.g. Ethernet/IP on rack 89) and the robot. You can map as few or as many bits as your device or connection supports.

It's up to you to lay out the I/O in a logical fashion so that you can configure and use it later.

Practical Example

Let's do a practical example where we setup an Ethernet/IP connection and map some Digital I/O (`DI` and `DO`) signals, our UOPs (`UI` and `UO`) and a couple Group inputs and outputs (`GI` and `GO`).

Most people start with a spreadsheet when planning out their I/O map. Let's plan on sharing 64 bits of I/O between our PLC and robot. Here's an example:

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

PLC Address	DI	Comment		DO	Comment
0.0	1	*IMSTP		1	CMD ENABLED
0.1	2	*HOLD		2	SYSTEM READY
0.2	3	*SFSPD		3	PRG RUNNING
0.3	4	CYCLE STOP		4	PRG PAUSED
0.4	5	FAULT RESET		5	MOTION HELD
0.5	6	START		6	FAULT
0.6	7	HOME		7	AT PERCH
0.7	8	ENABLE		8	TP ENABLED
1.0	9	RSR1/PNS1/STYLE1		9	BATT ALARM
1.1	10	RSR2/PNS2/STYLE2		10	BUSY
1.2	11	RSR3/PNS3/STYLE3		11	ACK1/SNO1
1.3	12	RSR4/PNS4/STYLE4		12	ACK2/SNO2
1.4	13	RSR5/PNS5/STYLE5		13	ACK3/SNO3
1.5	14	RSR6/PNS6/STYLE6		14	ACK4/SNO4
1.6	15	RSR7/PNS7/STYLE7		15	ACK5/SNO5
1.7	16	RSR8/PNS8/STYLE8		16	ACK6/SNO6
2.0	17	PNS STROBE		17	ACK7/SNO7
2.1	18	PROD START		18	ACK8/SNO8
2.2	19			19	SNACK
2.3	20			20	RESERVED
2.4	21			21	
2.5	22			22	
2.6	23			23	
2.7	24			24	
3.0	25	GI[1] bit 0		25	GO[1] bit 0
3.1	26	GI[1] bit 1		26	GO[1] bit 1
3.2	27	GI[1] bit 2		27	GO[1] bit 2
3.3	28	GI[1] bit 3		28	GO[1] bit 3
3.4	29	GI[2] bit 0		29	GO[1] bit 4
3.5	30	GI[2] bit 1		30	GO[1] bit 5
3.6	31	GI[2] bit 2		31	GO[1] bit 6
3.7	32	GI[2] bit 3		32	GO[1] bit 7
4.0	33			33	

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

I have columns for PLC address (the format is plcByte.plcBit), FANUC `DI` and `DO` index and fields for comments. I've added borders to group each byte together. This will make things easier if you eventually want to send numbers over group I/O.

> ***REMEMBER:*** *A* ***byte*** *is a contiguous group of 8 bits. A* ***word*** *is a group of two bytes (16 bits).*

We're going to use the first 3 bytes for our UOP signals. "But doesn't the spreadsheet say `DI` and `DO`?" you ask. Yes, but we can map the same bits to both places. When the PLC turns on it's output `0.0`, both `DI[1]` and `UI[1]` will turn `ON` on the robot side.

> ***NOTE:*** *We could just have easily started mapping our* `DI` *and* `DO` *signals at the 5th byte, avoiding any overlap with the UOP and Group signals (and I generally do!), but let's just map our entire connection to DI/DO and see what happens.*

Setting up an Ethernet/IP Connection

I won't go through the PLC-side of the Ethernet/IP connection setup (check the Ethernet/IP Manual), but it's very easy on the robot side. Make sure you've setup your IP address (`SETUP > F1 (TYPE) > Host Comm > TCP/IP`) and have the robot and PLC connected to your ethernet switch. Verify you can ping both devices from a computer on the same subnet.

Open up the Ethernet/IP connection list on the robot via `MENU > I/O > F1 (TYPE) > Ethernet/IP`. You'll see a list of connections. Cursor down to an unused connection and configure a new adapter (the robot is almost always the adapter in these connections) via `F4 Config`.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

All you have to specify is the number of words (groups of 16 bits) to be shared. We want 64 inputs and 64 outputs which is four 16-bit words on both the input and output size. The default alarm severity of `WARN` should be fine.

If everything's setup on the PLC-side, you should just be able to get back to the list with the `PREV` key, cursor over to the `Enable` column and use `F4 TRUE` to enable the connection. You'll see a `RUNNING` status as soon as the connection is successfully established.

> ***NOTE:*** *There aren't many places to make errors here, but I've seen issues when people have discrepancies between the PLC and the robot (e.g. PLC connection is setup for 128 bits where the robot is only setup for 64). Double-check that the devices can communicate with each other with a* `PING` *or two and make sure the quantity of I/O you want to exchange matches on both devices.*

OK. We have an Ethernet/IP connection, but we aren't using any of that I/O yet. Let's map some things on the robot side so we can make use of our connection.

I/O Configuration

Head to the Digital I/O setup page via `I/O > F1 (TYPE) > Digital`. You should see a list of unmapped points (as indicated by the asterisks). Hit `F2 CONFIG` to bring up the configuration page.

> ***NOTE:*** *FANUC robots start numbering at 1 while your PLC will probably start numbering at 0.*

Let's map all 64 input and output bits of our Ethernet/IP connection to some Digital I/O signals. Create a range of DI/DO from 1-64 then cursor over and enter Rack 89 (for

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Ethernet/IP), Slot 1 (for the first connection in our list of Ethernet/IP connections), starting at bit 1, the very first bit of our mapping. Flip over to the other side (inputs, if you just mapped the outputs or outputs, if you just did the inputs) via `F3 IN/OUT` and create the same entry.

Making changes to the I/O map requires a power cycle so you won't see anything on the `F2 MONITOR` screen until you do so. Toggle the breaker or Cycle Power from the Function Menu. When the robot comes back online you should be able to toggle some outputs on the robot side, verify that you see them on the PLC (e.g. turning on `DO[9]` turns on PLC input 1.0) and vice-versa (e.g. turning on PLC output `1.0` turns on `DI[9]`).

Mapping UOP I/O signals

We can map UOPs just like we mapped our Digital signals; we just connect fewer bits. `I/O > F1 (TYPE) > UOP` and then `F2 (CONFIG)`. The input range is 1-18, Rack 89 (for Ethernet/IP), Slot 1 (for our first Ethernet/IP connection) starting at point 1 (because that's what we defined in our spreadsheet).

The outputs have a slightly expanded range of 1-20 but require the same Rack, Slot and Start point.

After a power cycle you should be able to see some of the UOP output signals turn `ON` and `OFF` when you turn the Teach Pendant on and off. Flip over to the Digital Outputs (`I/O > F1 (TYPE) > Digital`) and see the same bits turn `ON` and `OFF` since these bits are actually overlapping on the connection.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

NOTE: *I personally don't like overlapping UOP signals with Digital signals because it can be confusing. Having the entire connection mapped will be helpful to illustrate how the Group I/O signals work, but we'll clean things up later.*

Mapping Group I/O Signals

Our spreadsheet defines two four-bit Group Inputs, `GI[1]` and `GI[2]` and an eight-bit Group Output `GO[1]`. Bear with me as we quickly review how binary numbers work. Feel free to skip this section if you're already familiar with them.

–

Quick Review of Binary Numbers

We usually use base-ten for our numbers, meaning that each digit represents that quantity of `10^x` where `x` is the position (starting at 0) of the digit. (e.g. `147 = 1*10^2 + 4*10^1 + 7*10^0 = 1*100 + 4*10 + 7*1`)

Binary (or base-two) numbers are similar, but we only get two choices for each digit: `0` and `1`, and we use `2^x` as the multiplier. The binary number `1011 (base-two)` works out to be `1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 1*8 + 0*4 + 1*2 + 1*1 = 11 (base-ten)`. It takes more digits to represent the same number in binary vs. base-ten, but it's convenient for our purposes where we want to send data with simple `ON` and `OFF` signals.

–

Bring up the Group Inputs via `I/O > F1 (TYPE) > Group`. Make sure you're looking at the Inputs (`GI`), toggling if necessary with the `F3 IN/OUT` softkey if necessary.

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

Now bring up the Configuration screen with `F2 (CONFIG)`. You'll see that this screen is slightly different from the Digital and UOP configuration screens, but you can probably guess how to setup the table.

Our spreadsheet shows `GI[1]` as a 4-bit number starting at the 25th bit (starting from 1) of our connection. Setup the first entry on rack 89, slot 1, start point 25 with 4 points.

Create a similar entry for `GI[2]`, only this one starts at point 29.

Flip over the outputs and setup `GO[1]` on rack 89, slot 1, starting at point 25 with 8 points.

We'll be able to receive values between `0` and `15` on `GI[1]` and `GI[2]`, and we'll be able to send values between `0` and `255` on `GO[1]`.

Once you do a power cycle, you should be able to assign a value to `GO[1]` and see the corresponding bits (`DO[25]` through `DO[32]`) change accordingly, representing that number in binary. If you send a `0`, all those bits will be `OFF`. If you send a `255`, they'll all be `ON`.

It's up to you to configure things on the PLC-side to read and write those integer values. You may have to do some bit-masking and bit-shifting trickery for those 4-bit Group Inputs, and you may find it more convenient to just use 8-bits (or even 16-bits for a maximum value of `65535`) for all your Group I/O.

"How do I send negative numbers?"

You can't, but you can decide on some fixed offset value to send and receive on the PLC-side and compensate similarly on the robot-side.

For example, let's say we mapped an 8-bit value which gives us `0` to `255`. If we wanted an even split, we could use an offset of `128` and interpret the result as `x-128`

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006

where `x` is the value sent or received. (e.g. PLC sends `255`, robot reads `R[1:adjustedGIValue]=GI[x]-128=255-128=127`. If the PLC sent 0, the robot would interpret `0-128=(-128)`.

"How do I send real or floating-point numbers?"

In a similar way, you'll want to send and receive values with some agreed-upon multiple (e.g. 100 or 1000, depending on the precision you want. If you had a 16-bit mapping, you could have the PLC send the value multiplied by `1000`, then you'll divide by `1000` on the robot side. (e.g. `R[1:adjustedGIValue]=GI[x]/1000 ;` or `GO[x]=(R[2:valueToSend]*1000) ;`)

Wrap-Up

I think that about covers it when it comes to mapping I/O.

You may find it useful to know that all those Flag signals are mapped to rack 34. If you mapped your UOP Input signals to rack 34, you could control the robot in remote via those Flags.

Another useful rack to know is rack 35 which gives you always `ON` and always `OFF` signals. Use slot 0 for `OFF` and slot 1 for `ON`.

Check the DCS manual for all the different things you can map on rack 36… lots of useful stuff there (e.g. Cartesian Position Check zone safe/not-safe signals, safe I/O, etc.).

Let me know if you have any questions!

Purchased by Niccolò Rorato, nickrorato@gmail.com #15955006