Programing a RCT

Introduction

OK, you've got your new RCT setup and the boss is making noises about seeing it do something more than run the demo program.

You've looked over the RCT Manual that came with the thing and (hopefully) found it an informative reference document. But it doesn't do much to show you how to write a real, useful program.

This document is supposed to be more of an introductory “how-to” for programming your Gen II RCT.

Although we can assume that some folks reading this will have a solid software background, and others will already have mastered the previous generation RCT, we're guessing that most will have some idea of what programming is about but no recent experience. We'll try to be useful to all.

This manual will attempt to guide you through the basics of getting a working program into your RCT.

For the purposes of this write-up we'll consider a mythical system with a load station, an unload station, and three cleaning stations. This may not perfectly match your system but it should give you an idea of how things work.

Each RCT system is made to match a specific cleaning system. And cleaning systems are custom packages.

This means that each system will have a different number of stations and the stations will spaced differently. So our examples will be for a mythical system with nice, neat numbers for station locations. Your system will undoubtedly be different and will require some customization.

The principles don't change but the specifics do.

So what's a program?

In the context of our RCT world, a program is a list of low level instructions that tell the RCT what to do. (GO_X, GO_Y, etc.)

Writing the program is the process of matching the available instructions with the task at hand.

If you had to give a not-so-bright worker a list of steps describing how to run baskets of product through a cleaning process, the list would be a program – and very similar to

the programs you'll be writing.

Your boss probably told you something like, “I want the RCT to grab a basket from the load station and carry it through cleaning stations 1,2 and 3, and then put it in the unload station”.

“In station #1 I want the basket of parts to be ultrasonically washed for 3 minutes. In station #2 I want the basket to be rinsed for 4 minutes, and in station #3 I want the basket to be dried for 5 minutes.”

And having said that your boss would assume he had fully defined everything and will be back in about 10 minutes expecting to see the program actually running. (After all, he's done the hard part by fully defining the cleaning process – all you have to do is key it into the RCT.)

When we look at what he's said (and has not said) we can better define the process.

Go to the load station.
Pick up the basket at the load station. (Maneuver the hook under the basket handle and lift.)
Go into cleaning station #1.
Turn ON the ultrasonics.
Wait three minutes.
Turn OFF the ultrasonics.
Pick up the basket.
Go into cleaning station #2.
Wait four minutes.
Pick up the basket.
Go into cleaning station #3.
Turn ON the dryer.
Wait five minutes.
Turn OFF the dryer.
Pick up the basket.
Move to the unload station.
Put down the basket in the unload station (disengage the trolly's carrying hook from the basket handle).

Looks like a whole lot more than what he said.

He was assuming that you're not an idiot (regardless of what he says) and that you would know that what he meant was not exactly what he said. If he is not an idiot.

In the listing above we've cheated a little bit by not going into the details of how to pickup a basket and move it to another location.

We're also simplifying our program by assuming that we can get away with a pretty

basic program, a program that picks up a basket and carries that one basket all the way through the system before it goes back to get another basket. The alternative would be to bring in a stream of baskets and move them all through the system in a continuous flow, having all cleaning stations in play at once. That's little harder to do, so we'll tackle that later.

When building our program we first have to find out “where” the locations in our system are in terms of the “counts” the system uses.

If you've read the beginning of the RCT Reference Manual you already know that the RCT uses incremental encoders that produce a pulse each time the RCT moves a fixed distance (about 3/8” or 10mm). The controller counts up the pulses to know where the RCT is. Of course this all assumes that you start counting from a fixed starting location. HOME is that location – all the way UP and all the way OVER.

So to define our stations we'll manually move (jog) the RCT trolly into the positions we'll want it to move to automatically; and we'll write down the locations to put into the program.

But first we have to HOME the system to be sure that the encoders are calibrated – that is, start counting from the “zero” location.

So lets start with the RCT ON and begin on the MAIN SCREEN. (Which is the screen that always comes up when you turn your RCT ON.)

Press the big, green *SYSTEM ENABLE* button on the control box to enable motion. (This is not an on-screen button. The SYSTEM ENABLE button is the green physical button on front of the control box, just below the touch screen.)

Move from the MAIN SCREEN to the MANUAL SCREEN by pressing the *Manual* button on the touch screen. (Top row, right side.)

Once on the MANUAL SCREEN you'll see a cluster of UP/DN/RT/LT jog buttons with a HOME button in the center.

Below the jog cluster you will find an X=xxxx, and a Y=yyyy position indicator (located between a HOME_X and a HOME_Y button).

Note:
HOME_X and HOME_Y are not available as program commands. You only get them on the manual screen.

HOME_Y moves vertically up to the Y home location.

HOME_X moves horizontally to the X home location. Be aware that if you are not at the Y top you run the risk of running into the side of a tank.

The standard HOME command that you can use in a program consists of a HOME_Y followed by a HOME_X command that are automatically combined in the RCT.

When you use a GO_X (0) or GO_Y (0) the RCT uses the opportunity to recalibrate the zero on the whichever axis is involved by automatically substituting an internal HOME_X or HOME_Y command for the GO_X(0) or GO_Y(0).

Make sure the RCT is clear for motion. (Nobody and nothing in the way of going HOME).

Press the HOME button. The RCT should respond by moving UP to the top (Y home) first and then over to the end (X home).

Note:
Yes, that X home location description is ambiguous; most systems have X home to the left. But some have the X home to the right. It's a customer choice at the time of purchase. So when referencing it here we're trying not to offend.

The HOME location is almost always above the LOAD station.

Once the HOME has been performed, hang a basket on the hooks and JOG over to the center of the first tank.

Then JOG down to the point where the basket is resting on the bottom of the tank and the hooks are completely below the basket's handle. (Disengaged.) Be sufficiently far down that if the hooks are moved towards HOME they will not touch the basket but not so far down that they will hit the rim of the basket if moved away from the handle.

Note:
If you have ultrasonic immersibles in your tanks instead of having transducers bonded directly to the bottom of the tanks, there will be racks in the tanks that allow the basket to sit just above the immersibles without actually touching the immersibles.

By now you've noticed that the JOG controls are not the most responsive things in the world. We agree. The slowness of the response is a function of the time required for the touch screen to communicate with the PLC that is actually controlling the motion. Just think how sluggish it would be if we weren't communicating via high speed ethernet.

It usually takes a few tries to get the feel for releasing the buttons a little bit before the basket gets to your intended location, but it can be done.

Now write down the X and Y locations shown in the position indicator box below the jog cluster. These represent the location of your STATION #1.

To test this out we can use the manual GO_X and manual GO_Y commands. These

commands are available on the MANUAL SCREEN.

First press HOME to move the basket back to the starting location.

Then press the blue indicator/button below the GO_X button on the left hand side of the screen. Key in the X value just measured for STATION #1 and press the enter key to complete the entry. (The OIT uses a graphic symbol "↵" for ENTER, just like the one on a PC keyboard.)

Note:
When we say "indicator/button" we're referring to an indicator (a window that shows some information) that may be pressed to key in a new value.

Some indicators are just indicators. We usually show them as "flat" areas with numbers within.

Some indicators are also buttons. We usually show those as buttons ("3-D" beveled edges on the buttons as opposed to "flat" indicators) with numeric indicators on the face of the button.

The indicator/button should now show the destination you just keyed in.

Be ready to press the red SYSTEM DISABLE (or E-STOP) in the event that something goes astray. (You never know.)

Note:
The SYSTEM DISABLE button is on the front panel of the RCT below the touch screen.

Now press the blue GO_X button. The RCT trolly should move over to a position directly above STATION #1 if the "jog" measurement was correct.

And now press the blue indicator button below the GO_Y button. Enter the Y-coordinate (vertical) the same way as you did the X-coordinate.

Press the blue GO_Y button. The RCT trolly should lower the basket into STATION #1, stopping with the basket resting on the bottom of the tank and with top of the hooks fully below the basket handle but above the rim of the basket.

OK now we know where STATION #1 is.

Do the same for all of the stations in your system. Don't forget the LOAD and UNLOAD stations.

I'll wait...

Now, for the purposes of this manual we'll assign some easy-to-remember locations to our idealized machine so there are sample numbers to put into the instructions in these examples (since there is no way to know where your stations are from here).

	X location	Y location
Load	30	60
Station #1	100	70
Station #2	200	70
Station #3	300	70
Unload	400	60

Notice that most of the Y locations are the same. This is common for RCTs. Tanks are generally designed to have the same bottom heights, wherever that is practical. And the spacing between tanks is most often a constant.

Also, we'll specify that:

- PER output #1 controls the ultrasonic generator in station #1
- PER output #2 controls the dryer blower
- IN#1 reads the load station sensor (OFF = empty, ON = basket in station)
- IN#2 reads the unload station sensor (OFF = empty, ON = basket in station)

OK now lets start putting together the rudiments of a program.

From where I'm sitting there is no way to know if there is a program of any kind already in your system. Maybe a demo from the factory. Maybe somebody else had been working with the system. So the best thing to do before we go further is to look, and if there is a program, decide if maybe we want to save it.

Press the blue *BACK* button in the lower right corner of the screen. That will return you to what ever screen you just came from. In this case, MAIN, which is where we want to go.

Now press the *EDIT PROGRAM* button. Second row, left-most button.

Did the RCT just say something rude? Maybe you haven't logged in as supervisor. Only a supervisor can work with a program. (Unless somebody ordered your system with security defeated.)

Note:
When the RCT is delivered (if there were no special instructions at the time of purchase) the supervisory USERNAME is “SUPER” and the supervisory PASSWORD is “super”.

USERNAMEs and PASSWORDs are case sensitive – “ABC” is not the same as “abc”.

There is also an operator's login, USERNAME = OPERATOR, PASSWORD = operator.

Yeah, not exactly inspired choices, but you're going to change them anyhow, right? How to change is in the “RCT Manual”.

It might be a good idea to change the default passwords at some point. The procedure

for changing the password is in the RCT MANUAL.

OK, now that we're in the EDIT PROGRAM screen, lets see if there is a program already in there.

The second indicator window from the left in the black horizontal bar (there's a label “SIZE” above the window) shows how many steps are in the CURRENT program. A CLEARed (erased) program has 1 step in it, and that step is NO-OP (no operation – a NULL instruction).

If the program has more than one step maybe you want to save the program? (Maybe not if you know it's not an important program.)

To save the CURRENT program you'll have to return to the MAIN MENU (press the blue BACK button in the lower right corner).

NOTE:
When we refer to the CURRENT program we're talking about the program that you can edit, save, run...etc. As opposed to any of the four programs that are stored in the memory locations.

This works a little differently than Gen I RCTs. In the older systems you would directly edit and run programs stored in memory locations. There was no need for SAVE and RECALL, just SELECT.

In the new Gen II RCTs you must recall a program from memory to the CURRENT program in order to edit/run the stored program. And you must explicitly save edited versions of the program if you want to retain them in a memory location.

The new system offers more flexibility. With a Gen II you may recall a program from memory, edit it to test some new ideas and then discard the modifications if you don't find them suitable without losing the copy already in memory. Or save the modifications in a different memory location, keeping both the old and new versions (in different memory locations). Gen I RCTs do not allow this.

Now select the SAVE / RECALL button on the top row to change screens.

And now we're in the SAVE / RECALL screen.

Use the ^/v selector arrows to pick a memory location in which to save the CURRENT program. Alternatively you may press the indicator/button located between the ^/v arrows and enter the number of the memory location. (Allowable range being 1..4).

Next, press the *ENABLE* key. The gray “*Save Pgm*” and “*Recall Pgm*” buttons will change to BLUE to show that they are ENABLED and ready for use. (Gray-ed buttons are showing that they are not available right now. We don't want people to accidentally SAVE or RECALL as that might overwrite an important program.)

Press “*Save Pgm*”.

Done.

Return to the MAIN MENU screen. (The blue *BACK* button again.)

Now let me put together a crude program with the faux locations. When you enter YOUR version of the program into the RCT substitute your X and Y locations for the generic ones in this document.

And if you have more or fewer locations you will have to improvise a bit.

Note:
By the way, when we writeup a program for a customer we usually put it into a spreadsheet.

The columns force structure that makes reading the program easier. It's a form-factor that most people can easily display and edit.

We almost always put in descriptive comments for each line of code so somebody else has some hope of understanding our intentions.

Generally we'll include a header that defines input assignments, output assignments, flag usage, and any other system specific information that somebody needs to understand program operation.

Instruction	***Arg1***	***Arg2***	***Comment***
HOME			Go to the HOME location.
GO_Y		60	Go down so the hook is below the basket's handle height.
GO_X		30	Move the hook under the basket handle.
GO_Y		0	Move up to the TOP. (Grabbing the basket in the process)
GO_X		100	Move over to the center of station #1.
GO_Y		70	Move down into station #1 so the basket rests on the bottom and the hooks are fully below the handle.
PER_ON		1	Turn ON the ultrasonics in station #1.
TIMER		180	Wait for 180 seconds (3 minutes) to pass.
PER_OFF		1	Turn OFF the ultrasonics in station #1.
GO_Y		0	Back up to the top, grabbing the basket with the hooks.
GO_X		200	Move over to the center of station #2.
GO_Y 70			Move down into station #2 so the basket rests on the bottom and the hooks are fully below the handle.
TIMER		240	Wait for 240 seconds (4 minutes) to pass.
GO_Y		0	Move up to the top, grabbing the basket with the hooks.
GO_X		300	Move over to the center of station #3.
GO_Y 70			Move down into station #3 so the basket rests on the bottom and the hook is fully below the handle.
PER_ON		2	Turn ON the dryer.
TIMER		300	Wait for 300 seconds (5 minutes) to pass.
PER_OFF		2	Turn OFF the dryer.
GO_Y		0	Move up to the top, grabbing the basket on the way.

GO_X	400	Move over to the center of the UNLOAD station.
GO_Y	60	Move down into the UNLOAD station so the basket rests on the bottom and the hooks are fully below the handle.
GO_X	380	Move the hook away from the handle.
HOME		Move up to the top and over to the LOAD station so we're ready to start again when the START button on the RUN screen is pressed.

And that is pretty much the simplest program that does the job. Of course we've taken a few shortcuts.

We don't check the LOAD station sensor to see if there is a basket waiting to be picked up. We simply assume that there is a basket there to pick up. If there isn't a basket present, we'll be moving a phantom basket through the system.

We don't check the UNLOAD station sensor to see if somebody has left a basket there. (If they have left a basket there, we'll have a crash when we try to drop a basket on a basket.)

We've assumed that the power door above the dryer (if present on your system) has been left OPEN. If not there will be a crash as we try to drop the basket through the closed door.

A crash means that the basket runs into something solid with potentially bad result. Something gets bent or someone goes *OUCH*.

Oh, and when running a test program with the TIMER instruction… you can cheat the timer when testing, to save time. While the timer instruction is running there is a "CANCEL TIMER" button on the screen below the TIMER count-down display. Pressing the button kills the timer and proceeds to the next step in the program.

You also may have noticed that there is no END statement at the, um, end of the program. It is not needed. An RCT program will terminate when it completes the last instruction in the program (assuming that the last instruction does not command the program to loop back; but we'll deal with that later).

In this case the END statement is optional, and being lazy, we left it out. (There are situations, discussed later, where it is not optional.)

Entering a Program

Lets spend a minute or two entering a few of the instructions into a program, just to get the feel for things.

Starting on the MAIN MENU screen, press the *Enter Program* button (second row down,

left side).

The upper part of the screen shows you a step in the program. At the moment, this is probably the first step in an empty program:

- The *Program Number* indicator will tell you the (memory location) number of the last program saved to or recalled from memory. The allowable range is 1..4.
- The *Size* indicator tells you how many steps (total) are in the CURRENT program. The allowable range is 1..400. Note that you cannot have a zero length program. If you try, the system will insert a null (NO_OP) instruction.
- The *Step* indicator/button shows you which step within the program is being displayed. The range is 1..400.
- Pressing the *Step* indicator/button allows keying in another step number to display. (If you key in a step number outside the program, say “150” in a 100 step program, the entry will simply be ignored.)
- The ^/v buttons let you increment/decrement the *Step*. (These buttons will be ignored if you try to go outside the range of the current program.)
- The *Instruction* window gives the name of the instruction in the step you're currently viewing.
- The *Argument* window shows the argument (GO_X **123**) or argument**s** (If_In_On **1,2**) associated with the instruction. That is, if the instruction uses an argument. (HOME, for example, has no argument.)

Below the current instruction is a set of buttons for selecting the new instruction that you want to assemble and put into the program.

On the right hand side of the display are buttons to:

- INSERT an instruction (that you have already assembled) into the program. The new instruction will be inserted following the one being shown on the upper part of the screen. And then the screen will show the newly INSERTed instruction with updated step and arguments. You may not INSERT after NO-OP (you must REPLACE the NO-OP) and you may not INSERT after the 400th instruction (the highest instruction allowed). You may not INSERT an instruction if there already are 400 instructions in the program. The INSERT button will be grayed out when it can't be used.
- REPLACE the displayed instruction with another instruction. This overwrites the displayed instruction in the upper screen window with the new instruction that has been assembled in the lower screen windows. The REPLACEd instruction resides in the same STEP as the instruction it has replaced, so the displayed step number will remain unchanged.
- DELETE the instruction shown in the upper screen window. Note, there always must be at least one (1) instruction in a program. If you delete the last instruction in a program the system will automatically insert a “NO-OP” instruction. Even if you CLEAR (delete) the program, you will have one instruction – a NO-OP. And the NO-OP can only be REPLACEd not DELETED. (Actually, if the last

remaining instruction in a program is a NO-OP and you delete it, it is deleted – and immediately replaced with another NO-OP instruction. But then, who really cares?)

There are a couple of indicators on the right hand side of the screen.

One is a READY/BUSY indicator. When INSERTing or DELETing an instruction all of the instructions above the instruction you're inserting or deleting must be moved in memory (up for insert, or down for delete). During the process of moving the other instructions, the indicator will show BUSY and the screen will not allow any other *inserts, replaces,* or *deletes* until the indicator shows READY again.

The other indicator is a “Pgm Changed” indicator. This comes on when a program has been changed in some way after being recalled or saved. This indicator is intended to tell somebody that a program is no longer the same as it was when last recalled from (or saved to) memory – Ie. has somebody been editing the program? Copies of this indicator appear on other screens – such as RUN for the obvious reasons. (We don't want somebody to unknowingly run a non-standard program that may not conform to the official process recipe.)

OK now lets start keying a couple of the instructions in the example.

- Press the HOME button to select the HOME instruction. The large gray window to the right of the button will give a simple description of the instruction.
- Home has no arguments so there is nothing else to add. The instruction has been “assembled” and is ready to use.

We're starting with a blank program so we must *replace* the NO-OP with our new HOME. (INSERT is not an option when there is a NO-OP.) So we press REPLACE.

- The NO-OP instruction in the upper window is replaced with a HOME.
- The Pgm Changed indicator shows that the program has been changed.
- The HOME in the lower screen (instruction assembly area) remains unchanged.

And we're ready to do the next instruction.

- Press the GO_Y button. The large gray window to the right of the button will give a simple description of the instruction.
- Note that there is a prompt below the ARGUMENT button telling you the allowable range of the argument (0..9999). That's at the top of the descriptive window.
- Press the single ARGUMENT button at the top of the descriptive window to be allowed to enter the argument. An entry outside the allowable range will be rejected (try it, if you like). Our instruction requires an entry of “60”.
- Press the INSERT button to insert the “GO_Y 60” that was just assembled into the program, after the HOME instruction.
- The READY/BUSY indicator will show BUSY for a moment and then will return to READY.

- The display at the top of the screen will show that “GO_Y 60” is now instruction #2 in the program.

Now let's say that you inadvertently entered a 50 instead of a 60 in the GO_Y instruction.

No big deal.

The GO_Y instruction remains unchanged in the instruction assembly area (in the lower screen). Just hit the “ARGUMENT” button again and key in 60 to change the argument. Then use the REPLACE button to replace the erroneous entry with the correction.

And if you don't like the GO_Y and wanted to replace it with a PER_ON instruction, no problem. Press the PER_ON button to select the instruction. Press the ARGUMENT button to enter the argument (0..99) and then press the REPLACE button. Done.

If you had selected a “two argument” instruction to edit, such as the If_In_On instruction, you would be presented with two argument buttons at the top of the descriptive window. (And shown the allowable range for each. Plus a quick description of what each argument does.) In dual argument instructions the arguments may be entered in any order.

If you accidentally (or on purpose) enter an out of range argument a yellow banner will appear across the top of the screen. Simply acknowledge the error by pressing the blue “OK” button and proceed.

When you're done editing a program you can leave the *Enter Program* screen using the blue *BACK* button in the lower right hand corner. (As usual.)

Flow Control

So how do we have the program test to see if the LOAD and UNLOAD stations are occupied or not?

Conditional statements. Program instructions that test an INPUT or a FLAG and control the flow of the program based on the result of the test.

Flow?

Normally instructions are executed in the order that they appear in a program's listing. But some instructions allow the order to be conditionally (if a condition is met) or unconditionally (just do it) changed.

Conditional branch instructions:

- IF this condition occurs, go to (branch to) another location in the program and

execute the instructions found there. Otherwise continue to execute instructions in the normal order.

Unconditional branch:

- Always GOTO another location in the program and execute the instructions found there. Never continue on to the next instruction (unless the next instruction is where you're branching to – no, I don't know why you'd want do that).

Subroutines:

- Go to another location in a program and begin executing instructions there…
- But remember where in the program we came from...
- Continue executing instructions "there" until a RETURN instruction is encountered. Then return to the location that called the subroutine and continue executing instructions as if nothing untoward had happened.

For the moment, the Conditional Instructions are the ones of greatest interest. They let us test a condition and do something special if the condition is met.

We know that IF there is a basket in the UNLOAD station IN#2 will be ON. And IF there isn't a basket in the UNLOAD station IN#2 will be OFF.

With that we can use a conditional instruction to make the program act differently if there IS a basket in the unload station than we would if there IS NOT a basket in the unload station. Like not dropping a basket off on top of the basket that is already there. (Always an embarrassment.)

OK, now we need another tool to pull this all together. The LABEL instruction.

When we enter instructions into a program the RCT automatically assigns them a "line number". The line numbers are sequential 1,2,3,...400. Every instruction has a different line number as an identifier.

So if we wanted to reference a particular instruction we could do so by the line number, no?

Trick question.

A line number is a good reference to start with. But if we insert another instruction in the middle of the list, the RCT bumps up every line number above the new instruction by adding 1 to their line numbers. This makes room for the new instruction in the list. And it changes the line number of every instruction above the newly inserted one.

So if we inserted an instruction after line 50, it would become line #51. And the previous instruction at line #51 would become #52, and ….

And a branch to a line number above 50 would be wrong.

And yes, the RCT could look for branches that would be affected by such changes (insert/delete) in the program and re-assign the new line numbers to the arguments of branch instruction. But the result, while functional, would make the program tougher to read and understand. You could never know where a GOTO instruction was taking you by the line number.

So we take a more “elegant” solution. We introduce the LABEL.

A LABEL is an instruction that does nothing but identify a place in the program.

LABELs are numbered and the program will support up to 100 different labels to GOTO (or GOSUB).

If a program has instructions inserted or deleted, the *line number* of a LABEL will change, but the LABEL's identifier (ZZ) never changes. The LABEL identifier is the argument of the LABEL instruction.

When a program reaches a LABEL ZZ instruction it does…nothing. And moves on to the next instruction in the list.

But when a branch to a LABEL ZZ occurs, the next instruction to be executed will be the LABEL regardless of where the branch instruction is in the program and where the LABEL ZZ instruction is in the program.

Note:
It goes without saying (or should) that there can only be one LABEL ZZ in a program. The results of having two label instructions with the same identifying number are too horrible to consider. (Generally, the program will find the label that occurs closest to the beginning of the program and ignore any other.)

So lets make a change to our simple program to look for a basket in the unload station and prevent any possible crash.

We're starting with:

```
……..
PER_OFF         2       Turn OFF the dryer.
GO_Y            0       Move up to the top.
GO_X            400     Move over to the center of the UNLOAD station.
GO_Y            60      Move down into the UNLOAD station so the basket rests on
                        the bottom and the hook is fully below the handle.
GO_X            380     Move the hook away from the handle.
HOME                    Move up to the top and over to the LOAD station so we're
                        ready to start again when the START button on the RUN
                        screen is pressed.
```

And we change it to:

……...

PER_OFF		2	Turn OFF the dryer.
GO_Y		0	Move up to the top.
GO_X		400	Move over to the center of the UNLOAD station.
LABEL		***01***	***Identify location #01***
IF_IN_ON	***02***	***01***	***If input #2 is "ON "go to LABEL 01 else continue on...***
GO_Y		60	Move down into the UNLOAD station so the basket rests on the bottom and the hook is fully below the handle.
GO_X		380	Move the hook away from the handle.
HOME			Move up to the top and over to the LOAD station so we're ready to start again when the START button on the RUN screen is pressed.

So the revised program proceeds in the same way it had until it had before, but now, just before the RCT goes to drop off the basket in the unload station, it checks IN#2 to see if a basket is in the way.

If there is a basket in the UNLOAD station IN#2 will be ON. And if IN#2 is ON the program loops back to LABEL 01. And then the program moves down to the *IF_IN_ON 2,1* statement. And continues to loop until the basket is removed. And then...

If there isn't a basket in the UNLOAD station IN#2 will be OFF. And the program continues to execute normally with no looping.

So just adding a LABEL and a single conditional statement prevents a possible crash. That was worth the effort. (Crashes are *really* embarrassing.)

Before we go any further, lets consider a small change to the above.

Our machine operator may not realize that the RCT is waiting patiently for him to remove the basket from the unload station. All he knows is that the RCT apparently isn't doing anything. Must be broke. Or maybe it's a good excuse for a coffee break. ("The thing just stopped moving...")

So lets make one other little change.

The RCT has a beeper built in. And the program can turn it ON or OFF as needed.

The beeper is "connected" to FLAG #99. When we use a PER_ON 99 instruction to turn ON flag #99 we also turn ON the beeper. The flag (and the beeper) will remain ON until turned OFF, generally with the PER_OFF 99 instructions.

So lets use the beeper to wake-up the machine operator when there is a basket in the

way at the unload station.

We start with:

```
……..
PER_OFF          2      Turn OFF the dryer.
GO_Y             0      Move up to the top.
GO_X             400    Move over to the center of the UNLOAD station.
LABEL            01     Identify location #01
IF_IN_ON   02    01     If input #2 is ON go to LABEL 01 else continue on...
GO_Y             60     Move down into the UNLOAD station so the basket rests on
                        the bottom and the hook is fully below the handle.
GO_X             380    Move the hook away from the handle.
HOME                    Move up to the top and over to the LOAD station so we're
                        ready to start again when the START button on the RUN
                        screen is pressed.
```

And change it to:

```
……..
PER_OFF          2      Turn OFF the dryer.
GO_Y             0      Move up to the top.
GO_X             400    Move over to the center of the UNLOAD station.
LABEL            01     Identify location #01
PER_ON           99     Turn ON the beeper
IF_IN_ON   02    01     If input #2 is ON go to LABEL 01 else continue on…
PER_OFF          99     Turn OFF the beeper
GO_Y             60     Move down into the UNLOAD station so the basket rests on
                        the bottom and the hook is fully below the handle.
GO_X             380    Move the hook away from the handle.
HOME                    Move up to the top and over to the LOAD station so we're
                        ready to start again when the START button on the RUN
                        screen is pressed.
```

Now looking at the program you might say – hold on a minute, the thing will always beep as it comes towards the unload station. And the reply would have to be, “Yes, you're right. Mostly.”

If the unload station is unoccupied the beeper only stays on as long as it takes the RCT to execute the conditional statement. At very most you'll hear a short “chirp”. If that's a problem can tell them that the chirp is a warning that the RCT is going to drop something off at the unload station – that usually works.

But if the unload station is occupied, the beeper is on continuously until the obstructing basket in the unload station is removed.

There are more elegant ways to do this, but quick and dirty works too. And it is quick.

Lets try another change to the program that makes the RCT wait for a basket to be put into the load station before running.

HOME			Go to the HOME location.
LABEL		***02***	***Identify location #02***
IF_IN_OFF	***01***	***02***	***IF IN#1 is OFF (no basket) loop back to LABEL #02***
GO_Y		60	Go down so the hook is below the basket handle.
GO_X		30	Move the hook under the basket handle.
GO_Y		0	Move up to the TOP.
GO_X		100	Move over to the center of station #1.

So that was pretty easy too.

Now the RCT will sit there, patiently waiting until a basket is placed in the LOAD station before running the rest of the program.

We aren't “beeping” here because the basket may not be ready when the RCT is. So why needlessly annoy people. We could make it repeatedly “chirp”, which is less annoying. I'm sure you can see how to do that on your own.

Hint:

HOME		Go to the HOME location.	
LABEL	02	Identify location #02	
PER_OFF		***99***	***Beeper OFF***
TIMER		***01***	***One second***
PER_ON		***99***	***Beeper ON***
IF_IN_OFF	01	02	IF IN#1 is OFF (no basket) loop back to LABEL #02
PER_OFF		***99***	***Beeper OFF***
GO_Y		60	Go down so the hook is below the basket handle.
GO_X		30	Move the hook under the basket handle.
GO_Y		0	Move up to the TOP.
GO_X	100	Move over to the center of station #1.	

Subroutines

We referenced subroutines when we started talking about flow control instructions.

Subroutines are little programs within programs that can be called from any place in the main program by using the “GOSUB ZZ” command. (ZZ being the LABEL# of the start of the subroutine.)

Which sounds confusing if you're not already an experienced programmer but…

Lets say you could customize the instructions in your RCT.

You might want to make yourself a single instruction that drops a basket at the unload

station. Of course it would have to check to see if there was a basket in the UNLOAD station and do any other special stuff associated with dropping at UNLOAD and bundle all that functionality up in a single, special instruction.

OK, you can do that. With a subroutine. And it works out really nicely because if you have several places in the program that need to drop a basket at the UNLOAD station you don't have to put a complete copy of the unload stuff at each place in the program – just put a GOSUB ZZ. And if you need to make a change in how the drop at UNLOAD works you only have to edit one place in the program (the subroutine) not every place that calls the subroutine.

There are three things that are a little different about how you write a subroutine.

- The first instruction will be a “LABEL ZZ” so you have a place to “call” when you want to run the subroutine.
- The last instruction in a subroutine MUST be a RETURN instruction. RETURN tells the RCT to go back to the place in the program that called the subroutine and continue executing program steps from there.
- There should be NO WAY the program can get directly (that is, without a GOSUB) to the LABEL ZZ instruction that starts the subroutine. Often the subroutines are grouped together after an END instruction.

Note:
Yes, there can be active code after an END instruction. Executing an END instruction causes the program to terminate.

There may be multiple END instructions in a program.

There may be executable instructions beyond an end.

Note:
If code for a subroutine is allowed to run without being “called” by a GOSUB instruction everything will run correctly until the RETURN instruction is reached. That's where things get nasty.

If no other subroutine is active then an error will occur and the program will shut down.

If another subroutine is active then this subroutine will RETURN to that subroutine's calling location. Messy at best. Horrible at worst.

So our DROP AT UNLOAD subroutine would look like:

END		***End of the main program and beginning of the subroutine(s)***
LABEL	***03***	***Use a “GOSUB 03” to call this “DROP AT UNLOAD” subroutine from anyplace within the main program.***
GO_Y	0	Move up to the top.
GO_X	400	Move over to the center of the UNLOAD station.
LABEL	01	Identify location #01

IF_IN_ON	02	01	If input #2 is ON then go to LABEL 01, else continue...
GO_Y		60	Move down into the UNLOAD station so the basket rests on the bottom and the hook is fully below the handle.
GO_X		380	Move the hook away from the handle.
GO_Y		***0***	***Move up to the top.***
RETURN			***Were done here so lets go back to the calling program.***

Note:
You can have a LABEL within a subroutine. In fact, anything you can do outside of a subroutine can be done within a subroutine.

A subroutine can call another subroutine. The RCT supports subroutine calls up to 30 deep (30x sub calling sub, calling sub,…). More than 30 deep will cause an error.

And yes, a subroutine can call itself (heavy duty programmers call it “recursion”). *Don't do that.* Unless you're a very good programmer the odds are you'll cause an overflow error (more than calls 30 deep, remember?). And there really isn't a good reason to use recursion in a RCT. (But you can.)

When we (the RCT manufacturer) write programs we are heavy users of subroutines. Even when they aren't necessary subroutines can simplify program readability and make modifying programs easier.

Programs we write generally have a section with subroutines for PICKing and DROPping at each station in the system. Then a program becomes mostly a series of subroutine calls.

Symbolically our previous program becomes…

```
HOME
GOSUB (pick at load station)
GOSUB (drop at station1)
TIMER (120)
GOSUB (pick at station 1)
GOSUB (drop at station 2)
TIMER (180)
GOSUB (pick at station 2)
GOSUB (drop at station 3)
TIMER (300)
GOSUB (pick at station 3)
GOSUB (drop at unload station)
END

LABEL #pick at load station
RETURN

LABEL #pick at station 1
RETURN
```

LABEL #drop at station 1
RETURN

……..

Progressive Programs
When we refer to “Progressive Programs” we're not talking politics. We're referring to RCT programs that “progressively” move a steady flow of baskets through a system.

Doing this efficiently requires a bit more programming sophistication than a simple “pick it up and carry it through” type of program.

For example, you can't use TIMER instructions to time how long a basket stays in a station. When a TIMER is running, the RCT cannot move until the TIMER times out, and that interferes with our progressive motion.

We have to keep track of which tanks are occupied and which are not. This so we can avoid collisions and also avoid the wast-of-time that happens when we go to “pick” from unoccupied stations.

The first new concept we need is the FLAG.

FLAGS
Flags in RCT are equivalent to “Boolean variables” in fancier languages.

Flags are places where we can store ON/OFF (TRUE/FALSE) information – otherwise known as “variables”. Kind of like “points” in a PLC.

Flags can be set to ON by using the PER_ON instructions and set to OFF using the PER_OFF instruction, just like peripheral outputs:

- PER_ON 20 sets FLAG #20 to ON.
- PER_OFF 20 sets FLAG #20 to OFF.
- With the exception of FLAGS 13,14,15,and 16, all flags are automatically set to OFF when a program starts to run.
- Flag 99 is set automatically set to OFF when the system is turned OFF. (Recall that flag 99 is the beeper and we don't want to be able to leave the beeper on indefinitely.

Note:
Flags 13..16 are called retentive. They maintain their set values.

Normally, when a RCT program starts to RUN, the RCT automatically clears all FLAGS in the system to OFF – except the retentive flags. These retentive flags may therefor be used to pass information from

RUN to RUN.

For example, a flag that says that there were baskets in some stations when the RCT was last turned OFF, if retentive, could be used to have a subroutine remove all baskets from the system before running the main program the next time the program is RUN.

Flags can be tested using IF_IN_ON and IF_IN_OFF instructions, just like testing INPUTs.

If we turn ON a flag when we drop a basket in a station and turn the same flag OFF when we remove the basket from that station then other places in the program can test the flag at any time to see if there is a basket in the station.

Another tool we need for progressive programs is the TIMED_PER (*which per, how long)* instruction. The TIMED_PER is a fancy replacement for the TIMER instruction.

TIMED_PER
The TIMED_PER instruction is a special type of timer. It can work with all outputs and all flags.

When the program encounters a TIMED_PER instruction:

- It turns ON the flag or output referenced in the instruction.
- At the end of the time intervals specified in the instruction (nominally each interval is 30 seconds, but specials may be ordered) the referenced flag or output is turned OFF.

These TIMED_PER timers do NOT require that the entire RCT stop and wait for the timer to finish up. Regardless of what the RCT is doing when the timer completes, the specified output or flag will be turned OFF.

This can be really handy. You can write a program where a TIMED_PER instruction directly controls the ultrasonic generator in a station so that the generator doesn't run too long and damage parts in the tank.

You can use a TIMED_PER to control a flag that indicates that basket has been in a tank for the time required by the recipe so that the RCT can know to pick up the basket and move it to the next station in the recipe. (Flag ON means time not up yet.)

All in all, a useful instruction.

Note:
The TIMED_PER always turns ON an output or flag when executed and then, at the specified time interval later, it will turn OFF the same output or flag.

This will happen regardless of what any other instructions do to the specified output or flag in between.

It is NOT recommended that programs turn ON or OFF flags or outputs that are currently being controlled by a TIMED_PER instruction. The results are well defined but not usually desirable.

Now that we have all of our tools lined up and ready we can consider what it takes to do a progressive program.

When we write programs we usually assign two flags to each cleaning station (we're not including the LOAD and UNLOAD stations which have sensor inputs associated with them). We use one flag to show that the station is or is not occupied by a basket. We use the other flag (with a TIMED_PER instruction) to show that the basket is or is not done with the recipe-required time in the tank.

Using these two flags we can determine:

- If a basket has been in the station long enough for the recipe and is ready to be moved to its next location.
- If a station is empty so that it is safe to move a new basket into the station.
- If a station is empty so we don't have to try to move a basket that isn't there.

We also usually write a “Progress One” subroutine that is repetitively called to move the baskets to their next station.

The Progress One subroutine has to:

- Decide when to call PICK and DROP subroutines to move baskets.
- Make sure that baskets stay in stations for the specified periods of time.
- Try not to drop a basket into a station that is already occupied.
- Do not try to move “phantom” baskets from empty stations.
- Handle incidental functions like operator prompting BEEPS.

We'll show a simplified Progress One subroutine below.

We begin by specifying the FLAGs we're going to use to keep track of current system status.

First we'll specify that:

- Station #1 occupied flag is 31
- Station #2 occupied flag is 32
- Station #3 occupied flag is 33

(Note: See the pattern?)

Then we'll specify that:

- Station #1 timer flag is 21
- Station #2 timer flag is 22

- Station #3 timer flag is 23

(Note: The pattern continues. The flag numbers were selected to make it easy to remember which flag does what.)

In the example we will call on subroutines to PICK at a specified station and DROP at a specified station. We won't show the actual PICK and DROP subroutines in the interest of not writing a whole book. As the teacher says – an exercise for the interested student (hated them – the exercise not the teacher). Full sample programs are available from the vendor.

We will assume that a DROP subroutine turns ON the flag that indicates a basket in the station.

And we will assume PICK subroutine turns OFF the flag that indicates a basket in the station.

The code fragment below is a PROGRESS ONE subroutine. The entry point is LABEL 70. It is written for our imaginary machine with three cleaning stations (ultrasonic station, rinse station, and dry station) in addition to a LOAD and an UNLOAD station.

Every time this subroutine is called it checks to see what baskets are ready and able to move – and moves them. It is designed to be repetitively called to keep all the baskets moving through the system.

```
LABEL            70    Entry point for the PROGRESS ONE subroutine.
IF_IN_OFF  33    72    If the flag says no basket in Station#3 skip over the rest of
                             stuff for this station.
IF_IN_ON   23    72    If the flag says that the timer for this station is still running,
                             skip over the rest of the stuff for this station.
GOSUB                  Pick at dryer station (Station #3)
LABEL            71    Loop to here for the beeping.
PER_ON           99    Turn on the BEEPER.
IF_IN_ON   02    71    If UNLOAD sensor says “occupied” then loop back and keep
                       on beeping.
PER_OFF          99    Turn off BEEPER
GOSUB                  Drop at UNLOAD

LABEL            72    Next station instructions.
IF_IN_OFF  32    73    If there is no basket in Station #2 we're done here,
                       so goto LABEL 73
IF_IN_ON   33    73    If there is a basket in the next station, skip move for now.
IF_IN_ON   22    73    If the timer for this station is still running, skip move for now.
GOSUB                  Pick at station#2
TIMER            30    Stop everything and wait for a 30 second drip-off.
```

```
GOSUB                       Drop at station#3 (dryer station)
TIMED_PER   23    10        Start timer for dryer for 10 x 30 sec = 5 min (since we've just
                            dropped a basket there)
TIMED_PER   02    10        Start the timer for the dryer blower.

LABEL             73        Next station instructions
IF_IN_OFF   31    74        If there is no basket in Station#1 then we're done here,
                                  so goto LABEL 74
IF_IN_ON    32    74        If there is a basket in Station#2 we can't move yet so
                                  goto LABEL 74
IF_IN_ON    21    74        If the timer for this station is still running skip move for now.
GOSUB                       Pick basket at Station #1
GOSUB                       Drop basket at Station #2
TIMED_PER 22      08        Start Station#2 timer for 4 minutes.

LABEL             74        Next station instructions.
IF_IN_OFF   01    75        If there is no basket waiting in the load station, skip.
IF_IN_ON    32    75        If there is a basket in Station #2,skip this move.
GOSUB                       Pick at LOAD station
GOSUB                       Drop at Station#1
TIMED_PER 21      06        Start a 3 minute timer for Station#1.
LABEL             75
RETURN                      We've completed one full check so lets RETURN to the
                                  caller
```

Calling the above subroutine will check to see which baskets are ready and able to move and moves them.

Calling the above subroutine repeatedly will keep all of the baskets moving through the system.

Usually we setup a main program loop that, among other things, calls this subroutine each time it loops.

Tip:
We often designate a special flag to signify that the program is being run without any of the station timers. This makes testing programs much quicker since with the flag ON you don't have to wait for the full recipe times at each station.

And you don't have to make any big program changes to use this feature.

An IF_IN_ON statement that tests this flag can be placed just before each statement that turns on the station timers. If the flag is ON we jump over the statement that turns on the timer.

Then, when you want to test the program without the timers, just insert a PER_ON statement at the start of the program – and remember to remove it when done.

Debugging Programs
Almost without exception there is some little mistake hiding in every new program.

It can be a simple “typo” where an incorrect button was pressed when entering the program.

Or it can be a more serious problem hiding in the logic of the program.

Whatever the cause, it can be a nuisance to run down.

Since we write programs too, we put in some tools to make running down errors a little less frustrating.

There are always the traditional debugging tricks:

- You can put an END statement at various points in the program. When the END is reached it will stop the program and allow you to interrogate the state of the machine.
- You can use our WAIT statement to have the program stop and wait for you to acknowledge before proceeding instead of ENDing. This can help by keeping the program from moving so fast you can't follow what is happening.
- You can have the program set special flags to indicate when certain tasks have been completed.
- You can use our built-in debugging tools.

New with the Gen II version of the RCT are the following debug tools:

- Single step operation. The program runs ONE step of a program and then stops and waits until you acknowledge before performing the next step (and then waits…). This allows you to see the “flow” of the program at human speed.
- “No motion” operation. This tells the RCT to skip all motion related instructions. GO_X, GO_Y, HOME, GO_LOCN, and AGITATE will be skipped over. Using this tool you can speed up testing – you no longer have to wait while the trolly travels the length of your system. Can be combined with Step-Step operation.
- No stall operation. You can tell the stall-detection system to ignore stalls. This tool is only recommended for debugging motion (hardware) failures only. Be aware that the stall detect system is considered a ***safety feature***. Defeat this safety ONLY WITH EXTREME CAUTION. But it is available.

All of the above debugging modes can be turned ON and OFF from the UTILITY screen.

Needless to say, it is important to NEVER leave a system in any of the debug modes when a normal user/operator has access to the system. What the system will do will be outside their experience and as a result, what they do may be outside your experience.

The UTILITY screen is accessed from the MAIN MENU and requires a supervisor's password to access it.

The VIEW / CHANGE FLAGS screen is accessible from both the MAIN MENU and the RUN screen. If the user has a supervisor's password.

The VIEW / CHANGE FLAGS screen allows the viewing the state of all FLAGS and INPUTS, and manually setting the state of all FLAGS and OUTPUTS. This screen is accessible from the RUN screen and the MAIN MENU and can be used in conjunction with all other debug modes and features to follow program operation. The screen may be entered and exited while a program is running or after program termination.