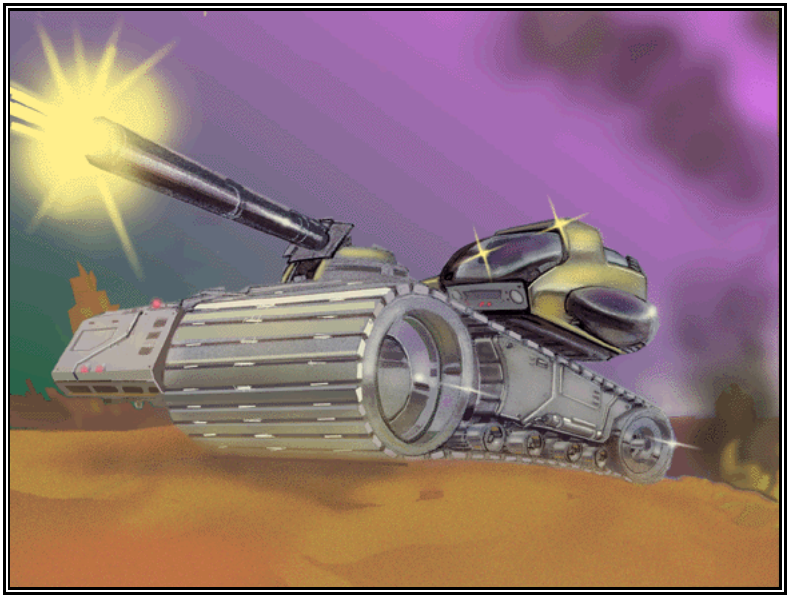
INTELLIBOTS™

... An Intelligent Programming Adventure



VERSION 1.0

Acknowledgements:

The producers would like to thank the following people, without whose invaluable help this production would not have been possible:

Julian Critchfield, Doug Earl, Mike Earl, Mark Hamilton, Col Jones, Matthew Leavitt, Harry Mahoney, Bob Taylor.

INTELLIGENT TECHNOLOGIES, INC.
P.O. BOX 2022
OREM, UTAH 84059-2022
U.S.A.

Table of Contents

Welcome to IntelliBots™

IntelliBots is an intelligent adventure in computer programming. Here are some important things you should know about IntelliBots:

- This is absolutely the most *fun* way to learn computer programming!
- IntelliBots is a *complete programming environment*, where you learn to design, create, and test you own computer programs using authentic, proven methods.
- IntelliBots helps you develop logic and problem-solving skills.
- IntelliBots is based on reasoning and logic, *not* on reflexes or hand-eye coordination.
- You can use IntelliBots by yourself, with friends, or in the classroom.

IntelliBots can teach you the basic skills you need to program popular types of computers, such as Macintosh and PC's.

Here are the topics in this section:

The IntelliBots Story

Humans had begun their inevitable migration into space and on one of the habitable planets that they colonized, they discovered technologies of an alien race long since departed. The technologies included new power sources, metal alloys, and other items yet to be understood. The most curious items discovered were large self propelled machines that appeared to have been designed to perform some form of automated labor. What could be understood of these new technologies was adopted and used to further the colonization effort on the planet.

After a period of time the planet was suddenly invaded. Who or what the invaders were was unknown since the colonists only saw their machines of war. The colonists quickly responded by mounting weapons on their own machines. Because of their limited population, the humans had decided to make the machines capable of operating on their own. This meant that these "intelligent robots" or IntelliBots must be programmed to perform their respective duties. The change in their roles also required a change in the way the machines were programmed.

You are about to enter the training school for potential IntelliBot "trainers" or programmers. You will learn how to program your IBot™ (as they have come to be called by the students) to move, to observe and learn about its surroundings, to defend itself, and to try to eliminate the alien invaders.

Getting Started

Important: Before you begin reading this IntelliBots Manual, take a few moments to review the *ReadMe* file on the installation diskette. It contains important information about IntelliBots, such as package contents, equipment needs, installation, and how to get help.

Here's what you should read to learn to use IntelliBots:

- *IntelliBots Manual* (what you are reading now) explains how IBots move and function, and how the IntelliBots software works. Then it teaches you important computer programming concepts so you can carry out many interesting and challenging programming tasks.
- *On-line Help*. Standard and custom help features are available for you in IntelliBots.
- *Programmer's Quick Reference* is the at-a-glance source of information for IntelliBots features such as instructions, ports, coprocessors, constants, and more.

How IntelliBots Works

As you read the IntelliBots Manual, here are the steps you will follow to write and run computer programs with IntelliBots. (If you're new to programming, don't worry; the manual will help you understand how to do these steps.)

1 You decide on a Mission (programming task) you want to accomplish.
2 In the Program Editor (like a word processor), you write a program (computer commands) to carry out the Mission.
3 You assemble the program (turn your commands into computer-readable instructions).
4 You set up the Mission by selecting the program you wrote and as many as three other programs to compete against it.
5 You select a Locale (a color map with terrain and objects) for your Mission. IntelliBots includes a variety of Locales, such as Desert, Meadow, City, Field, etc.
6 You run the Mission. Then IntelliBots makes these things happen:

- Each program makes an IBot move, turn, or do other tasks on a Locale on your computer screen.
- If you run several programs at a time, IntelliBots gives a small amount of time to each IBot in turn. The computer instructions run so quickly, it looks like all IBots are moving at the same time.
- The first IBot to accomplish the Mission wins, and statistics are displayed.

To win, you write the most effective program for the Mission. You can change your program and rerun the Mission as often as you want. You can also switch Locales and customize the way IntelliBots works. As you will see, IntelliBots is a fun, exciting, and *visual* way to learn computer programming.

IntelliBots Features

Here's a quick overview of important IntelliBots features you'll use. The technical terms used below are described later in the IntelliBots manual.

✧

✔ "SEEING" THE PROGRAM RESULTS

IntelliBots lets you *see*, on the computer screen, how your IBot follows the instructions in the computer program you wrote. You can write your own IBot programs to carry out Missions and compete with other IBots.

✧

✔ ACTION CONTROL

- You can change action speed, or pause or stop the action at any time.
- You can turn sound effects on or off.
- To see the final result statistics sooner, you can run a Mission without showing the IBots or Locale. The IBot programs run faster this way.
- You can use or disable the IBot offense (laser and shell commands) as you prefer.

✧

✔ STATUS BOXES

For each IBot program that runs, a status box displays on the Mission console screen. Each status box shows how well the IBot is doing on its Mission, displaying the IBot name and icon (picture), levels for the shields and armor and power, and status messages.

✧

✔ COMPETITION

The Competition feature lets you run your program against other IntelliBots programs to see which is the most effective. (The other programs can be written by your friends, or you can use the ones included on the IntelliBots program disks.) A Competition can be a Challenge, Round Robin, or a Race. You can run your programs with any of the provided Locales, and you can get detailed statistics for how well each IBot program did.

✧

✔ PROGRAM EDITOR

The IntelliBots program editor is where you write your programming source code. It's like a simple word processor, where you type, edit, find, and replace text. You can keep several editor windows open at a time.

✔ Built-in Assembler

The IntelliBots assembler translates your programming source code into code the IBot "understands." This assembler's instructions are like those used by today's Macintosh and PC computers. The assembler has many of the features found in today's top software development tools; it also includes directives, system commands, coprocessor commands, ports, and comprehensive error messages.

✔ Interactive Debugger

IntelliBots has a full-featured program Debugger that helps you:

- Display your program's instructions and data.
- Step through program instructions one at a time or continuously.
- Set and clear program breakpoints and condition flags.
- Change values in registers, ports, and memory, and monitor port data.
- Simulate program interrupts.

✔ Coprocessors, Ports, and Interrupts

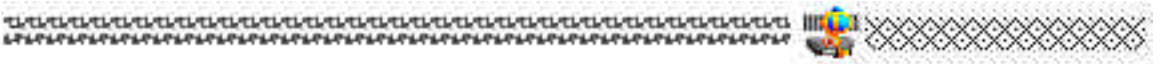
IntelliBots simulates the use of *coprocessors*, *ports*, and *interrupts*, helping you learn to program with these important parts of a computer in mind. IntelliBots uses coprocessors and ports so IBots can turn, move, scan, and so forth.

✔ File Management

IntelliBots includes important file management features. *New*, *Open*, *Close*, *Save*, and *Save As* to help you manage your program files. *Build* lets you assemble your source code into a runable program. *Generate Report* lets you display, and print statistics for one or more IBots. *Get IBot Info* shows statistics for a selected IBot. *Print* helps you manage the printing of your source code files.

✔ Easy-to-Read Manual

The IntelliBots Manual is designed to be easy to read and technically sound. Each lesson teaches you important concepts in computer programming, such as registers, loops, tables, and more. And be prepared to have a lot of fun while you're learning!



FAQ (Frequently Asked Questions)

Here are some of the most frequently asked questions about IntelliBots, along with our answers.

Q1 — How long does it take to learn how to program computers?
answer- Exactly five months and three weeks ... no; actually that depends on you. Completing the chapters in this manual will give you a well-rounded and fun education in programming. You can easily make the transition to other commercial programming languages.

Q2 — Is IntelliBots a game?
answer- IntelliBots is a professional-style programming environment with a game-like setting. When you run IntelliBots programs, you see them interact; this makes it a lot more fun to try out what you've learned. Still, you don't need video-game skills, such as hand-to-eye coordination, to succeed at IntelliBots.

Q3 — Do you have to know advanced math to learn programming?
answer- Actually, you can get started programming with just basic math – addition, subtraction, etc. Later, you will learn about base 2 and base 16 numbering systems. Logic and problem-solving skills are important in any programming, while advanced math is usually limited to specialized areas of programming.

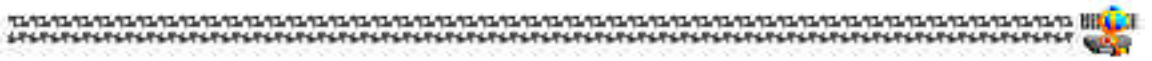
Q4 — Can IntelliBots be used in schools?
answer- Yes, and quite well. The Teacher's Guide includes curriculum guidelines and suggestions for classroom use. IntelliBots can be set up, administered, and run on a network for easy classroom use. And students can compete in in-class, class-to-class, and school-to-school easy-to-run tournaments.

Q5 — What skill levels does IntelliBots cover?
answer- IntelliBots covers many skill levels, whether you're a brand-new or experienced programmer. The manual contains many interesting exercises to teach you the fundamentals of programming and sharpen your skills.

Q6 — What kinds of careers are there in the software industry?
answer- Skilled, high-tech jobs include computer programming (writing programs for computers to run), software quality assurance (testing programs and systems), technical writing, systems analysis, management, and many others.

Q7 — What's it like being a professional programmer?
answer- Computer programming is one of the most in-demand careers in today's technology-hungry society. Programmers create software for a variety of uses, such as finance, global communications, education, entertainment, etc. And it usually pays pretty well.

Q8 — What good is IntelliBots if I don't want to be a programmer?
answer- IntelliBots can give you an understanding and appreciation of programming, which helps you develop logic and problem-solving skills that can help you in many areas of life.



What's Next ?

Look for these new products from Intelligent Technologies:

- Course 2: Intermediate Programming Concepts
- Course 3: Advanced Programming Concepts
- Course 4: Computer Mathematics
- Introduction to the C Programming Language

****

An intelligent, fun, and exciting computer programming adventure awaits you. To continue, turn to *Chapter 1 - About Computers and IBots* to learn more about computers and programming.

Computers, IBots, & Programs

In this section, you will learn about how computers operate internally, how IBots and IBot programs work, and how to change and assemble source code in programs.

- *Chapter 1 - About Computers and IBots* explains how you can combine IBot programs and Locales to create Missions.
- *Chapter 2 - Running IBot Programs* shows you how to select and run IBot programs, and how to control the on-screen action.
- *Chapter 3 - Changing IBot Programs* introduces the program editor and explains how to open, edit, and assemble source code files.

1 - About Computers and IBots

In this chapter you'll learn basic concepts about computers and programming. You'll also learn how IBots are like computers, how they move and function, and how they use programs to accomplish tasks. Here are the main topics in this chapter:

- About Computers
- About Computer Programming
- About IBots
- IBot Programs and Missions
- Locales
- Identifying Locale Objects

About Computers

A computer is an electronic device that lets you store, retrieve and manipulate information. Computers use *hardware* and *software* to accomplish tasks.

Computer Hardware

Hardware refers to actual parts of the computer, such as the keyboard, memory chips, monitor, disk drives, etc. Using IntelliBots helps you learn about these types of computer hardware:

- *CPU* (Central Processing Unit), like the "brain" of the computer. It sends commands to the other hardware pieces and controls the data being used.
- *Memory*, an electronic data storage area somewhat like human memory.
- *Registers*, very fast memory locations typically found inside the CPU.
- *Coprocessors*, specialized processors that perform specific tasks. These extra processors free up the main CPU so it can continue with other tasks.
- *Ports*, special memory locations that communicate with hardware devices.

The diagram below shows how these hardware items might be arranged in a typical computer.

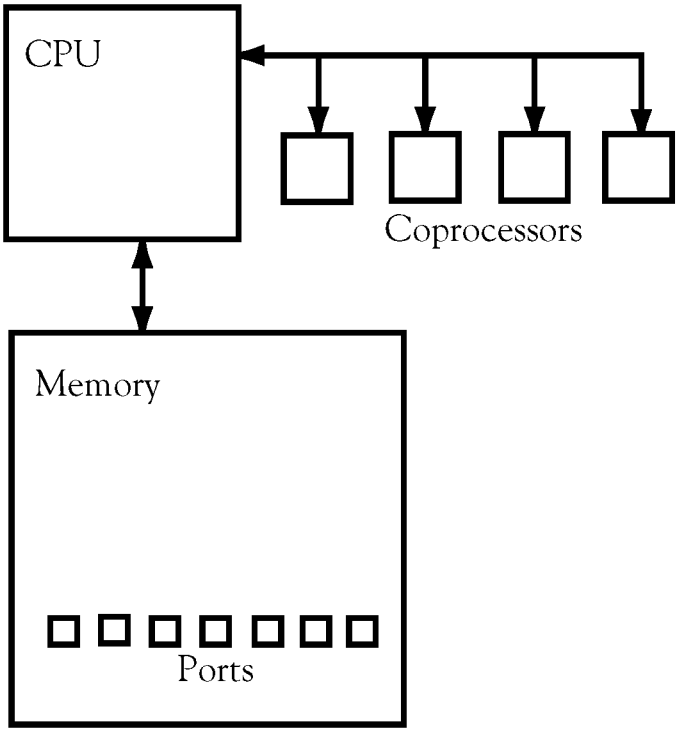


Figure 1-1: Computer hardware components

Computer Software

Software is one or more computer programs made up of an organized set of instructions that tells the CPU what to do. Software is usually created by a computer programmer and placed on a disk. The user of the software then loads it into the computer. When software runs, it is copied into the computer's memory, where it gives the computer instructions to perform certain tasks. Software contains the actual *machine code* (1's and 0's) that the CPU needs to use.

Examples of software include operating systems, word processors, spreadsheets, games, etc.

About Computer Programming

This section gives a brief introduction to computer programming. If you're new to programming, or if you have never programmed in an assembly language, you should read this section carefully.

What is Computer Programming?

Computer programming is the process of creating instructions for the computer's CPU to use. Here are the basic steps in creating a program:

1. Determine the objective you want to accomplish.
2. Design a plan to accomplish it, with simple English commands.
3. Convert the commands into *programming language* instructions and type them in a program editor.
4. Using an *assembler* or *compiler*, change the programming language instructions into actual machine code the computer can understand.
5. Run and test the program to make sure it solves the task correctly. (A program always does exactly what you *tell* it to do, but that may not be what you really *wanted* it to do.)
6. Once the program is in use, make and test changes to support the program's success.

These steps are important for *any* kind of computer programming, including IntelliBots.

What is a Programming Language?

A programming language, such as assembly, C, or Pascal, contains the symbols and rules for creating a computer program. (You can use English to write a book; you can use assembly to write a program.) The rules and symbols must be used correctly, or the program won't work.

After you write your instructions in the programming language, you use software called an *assembler* or *compiler* to convert them into machine code the computer can actually run. (The assembly process is explained in more detail in *Chapter 3: Changing and Assembling Programs*.)

About IBots

An IBot is an Intelligent roBot that looks and acts like a computerized vehicle on your monitor. To use IntelliBots effectively, you should understand how an IBot resembles an actual computer and how an IBot vehicle moves and functions.

The IBot Computer

The IBot "thinks" in a way that is modeled after an actual computer. It simulates the use of hardware, such as memory, registers, coprocessors, and ports. Each IBot is guided by a computer program that you, or someone else, creates.

The IBot Vehicle

The IBot vehicle has two parts: the *chassis* (the main body of the IBot) and the *turret* (where the laser and cannon are located). On the monitor, you see only a top view of each IBot in a Mission. The four IBot vehicles are shown below.

IBot #1 IBot #2 IBot #3 IBot #4

Figure 1-2: IBot vehicles

The next diagram shows the IBot in a 3-D view to help you see the parts and functions of the chassis and turret.

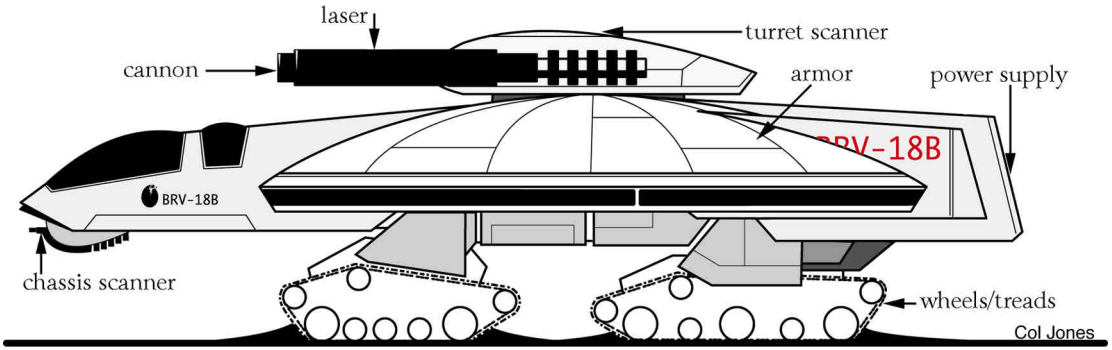


Figure 1-3: IBot chassis and turret, 3-D view

Chassis parts

- *Wheels/treads*. The chassis uses wheels and treads to turn and move in any direction, even backwards. When it moves diagonally, it travels at the same speed as it does when going vertically or horizontally. (IBots move only in two dimensions; they cannot fly.)
- *Chassis scanner*. IBots can detect low or high objects by scanning with the chassis. Objects and scanning are described in *Terrain* and *Scanning for Objects* later in this chapter.
- *Shield*. An IBot can boost shields, one for the front, back, left, and right sides. A shield is decreased when it is hit by a laser or bomb, or when the IBot runs into certain objects on that side.
- *Armor*. An IBot has four armor plates, one for the front, back, left, and right sides. When the shield on a side of an IBot is gone, the armor on that side is decreased each time that side is hit by a laser or bomb, or when the IBot runs into certain objects on that side.
- *Power supply*. An IBot begins with a full supply of power. When the armor on a side of an IBot is gone, the overall power supply is decreased each time that side is hit by a laser or bomb, or when the IBot runs into certain objects on that side. The power supply is also drained any time the IBot runs into a power object (see *Damage from Terrain Objects* below), and gradually as the IBot uses program instructions.

Turret parts

- *Turret scanner*. IBots can detect high (tall) objects by scanning with the turret. Objects and scanning are described in *Terrain* and *Scanning for Objects* later in this chapter.

- *Laser*. With the laser, the IBot can hit only high objects. At longer distances, the laser causes less damage unless you increase its power.
- *Cannon*. The cannon can hit high or low objects. Each shell radiates damage out from the point where it explodes.

The turret can turn in any direction, regardless of where the chassis is facing. IBots can hit or miss targets, and can move to avoid being hit. If an IBot is damaged so much that its power supply is gone, its program stops running.

Note: In Preferences, you can disable offense (laser/cannon) for all IBots.

IBot Programs and Missions

IntelliBots helps you create and test IBot programs, using the Editor, the Assembler, and the Debugger. You then run these programs to fulfill IBot Missions, which are tasks that IBots are designed to carry out. The Mission runs on a Locale, which may have several kinds of objects, including goals. Locales and goals are explained below.

In a Mission, each IBot tries to defeat other IBots, reach a goal, or complete a certain programming task. At the end of a Mission, the winning IBot is indicated, and statistics are displayed to show how well each IBot did in the Mission.

Locales

A *Locale* is the color map background you select for an IBot Mission. Although certain Missions work best with certain Locales, IntelliBots lets you run any IBot program with any Locale.

Each Locale is subdivided into a certain number of *units* (grid squares) that determine the size of the Locale and where objects are placed. The units don't actually appear on the Locale unless you use the Show Grid feature in the Mission Setup dialog.

Below is a picture of the `Desert2` Locale with several types of terrain identified.

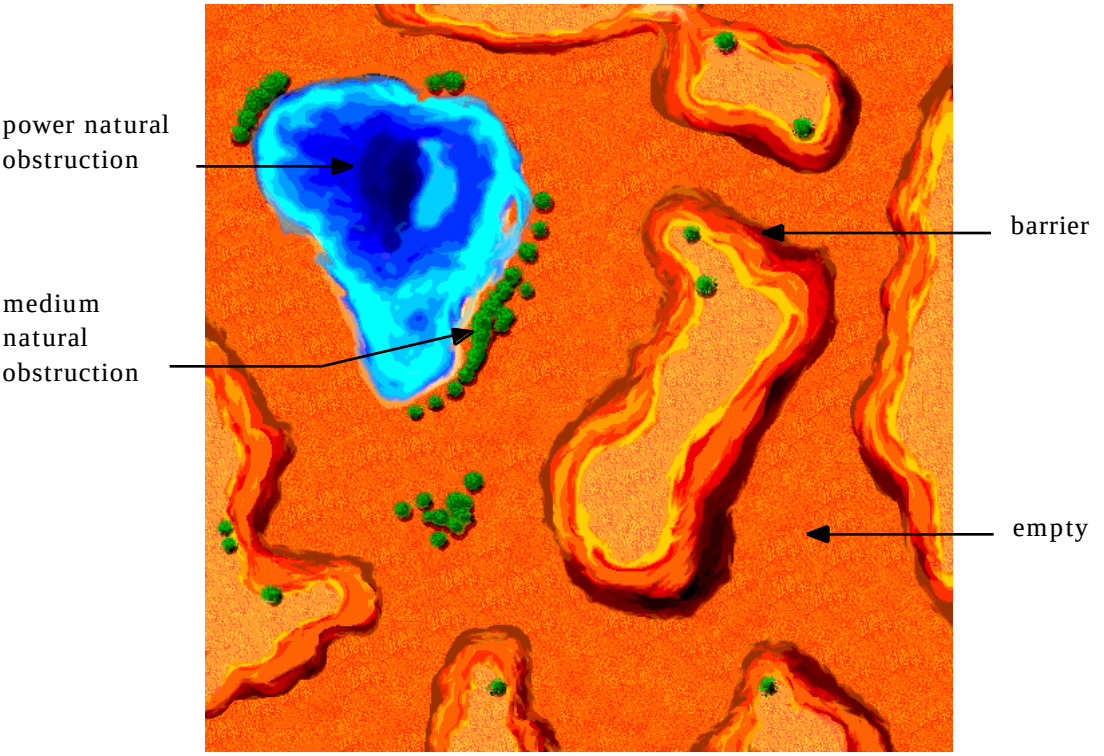


Figure 1-4: `Desert2` Locale

Terrain

Each Locale has different types of *terrain*. These are natural or artificial objects shown on the Locale, such as trees, rocks, water, and buildings. Terrain can be *low*, such as short grass or low walls, or *high*, such as trees or cliffs. For more details on terrain objects, see *Checking Object Types* in chapter 6: *Coprocessors and Ports*. You can also find complete damage descriptions in *Terrain Damage and Changes* in the *Programmer's Quick Reference*.

Damage from Terrain

When an IBot runs into certain kinds of terrain, it receives light, medium, heavy, or power damage. For example, an IBot that runs into a light object, such as a small tree, will be lightly damaged, and an IBot that sits in a swamp will continually have its power drained. IBots can also damage certain terrain objects by running into them or hitting them with laser fire or shells.

For more details on how IBots and terrain are damaged, see *Identifying Locale Objects* in *Chapter 2 - Running IBot Missions*.

Scanning for Objects

As explained before, an IBot can scan for objects with its chassis scanner or turret scanner. The chassis scanner finds both low and high objects; the turret scanner finds only high objects. The chassis scanner is a short-range scan. An IBot can also scan for other IBots on the Locale (IBots are defined as high objects).

With a turret scanner short-range scans (from 1 to about 15 Locale units) will identify objects accurately, but long-range scans (more than about 15 Locale units) sometimes won't give any useable information. Also, you can make an IBot temporarily invisible to turret scans by using "cloak" mode (as explained in *Chapter 7 - Computer Numbers and Bit Testing*).

Goals

Each Locale has zero, one, or more goals. If a Locale has a goal, it is not in effect for a Mission until you enable the goal. To enable it, you click Enable Objects in the Setup dialog box. You'll learn more about the Setup dialog feature in *Chapter 2 - Running an IBot Program*.

If the goal(s) is enabled, the first IBot that fires on it or runs into it wins the Mission. If yours is the only IBot in the Mission, and the Locale has no goal, you do not win the Mission; instead, the Mission continues until your IBot runs out of power, or until you stop the action.

IBot Placement on Locales

You can put one, two, three, or four IBots on any Locale. You can tell IntelliBots to use *fixed placement* or *random placement* of IBots. In fixed placement, the first IBot starts at location 1 on the Locale, the second starts at location 2, etc., and those locations are always the same for every Mission on that Locale. In random placement, the starting locations are chosen randomly, so the IBots will start each Mission on the Locale in locations that are more unpredictable. You will learn about setting IBot placement in *Chapter 2: Running an IBot Program*.

Whether fixed or random placement is used, each IBot is placed so it has a fair chance to find a goal or avoid damaging objects.

Identifying Locale Objects

An IBot identifies Locale objects by scanning them, but you can identify those objects yourself by using your system's pop-up Help, whether the Mission is paused or running. To identify Locale objects,

1 Choose pop-up Help from the Help menu.

2 Move the pointer over a Locale object.

Pop-up help appears over the object, describing the following information:

- The object's type
- Whether the object is high or low
- What kind of damage the object causes to an IBot that runs into it
- What the object converts to when hit by an IBot, laser, or shell.

Terrain Types

Each terrain type in a Locale is described below.

Low Terrain Type	Description (Examples)
Empty	Level terrain (dirt or grass)
Low cover	Ground-level cover (short vegetation or shrubs)
Light natural	(Small trees or rocks)
Light artificial	(Low walls or small objects)
Power trap, natural	(Sand, mud, swamp)
Power artificial	(Extensive rubble, debris)

High Terrain Type	Description (Examples)
High cover	(Tall plants, reeds, etc.)
Medium natural	(Forest or boulders)
Medium artificial	(Walls or small buildings)
Heavy natural	(Thick forest or large boulders)
Heavy artificial	(High walls or larger buildings)
Barrier	Can't be traveled over (impassable wall, cliff, etc.)

Summary

Concepts

These concepts were discussed in chapter 1: *About Computers and IBots*:

A) A computer uses hardware and software to carry out tasks.
B) Programming is the process of creating instructions for the computer.
C) Examples of programming languages are assembly, C, and Pascal.
D) IntelliBots programs are written to control IBots in Missions.
E) Locales are used as backgrounds for IBot Missions.
F) A Locale may contain a variety of obstacles and objects.
G) An IBot wins a Mission by reaching a goal or by defeating other IBots.
H) A goal is only in effect when it has been enabled in the Mission Setup.
I) IBots move in almost any direction on the Locale.
J) IBots can fire lasers or shells at other IBots, at goals, or at objects.
K) Chassis scans find high or low objects; turret scans find only high objects.

Terms to Know

After reading this chapter, you should be able to briefly define the terms below. If you need help, reread the chapter or see *Glossary* in this manual.

1) hardware; 2) CPU; 3) memory; 4) software; 5) machine code;
6) computer programming; 7) programming language; 8) assembler;
9) IBot; 10) chassis; 11) turret 12) scanning; 13) shield;
14) armor; 15) power supply; 16) laser; 17) cannon; 18) Mission;
19) Locale; 20) goal; 21) Locale units; 22) grid

2 - Running IBot Programs

In this chapter you'll learn how to run IBot programs in a mission. You'll see how IBots move and interact with Locale objects in a Mission. Here are the main topics in this chapter:

- Setting Up a Mission
- IBot Status Boxes
- Running a Mission
- View and Speed Options
- Running Competitions
- IBot Statistics

Setting Up a Mission

Before you run an IBot Mission, you need to select both a Locale and IBots, and do a few other setup tasks. Follow the steps below to learn how to set up and run a single Mission.

1 Choose Setup from the Mission menu or from the main IntelliBots dialog.

Important: Many of the steps in this IntelliBots Manual refer to using the menus, but you can select any of the same features from the main IntelliBots dialog as an alternate method.

The Mission Setup dialog box appears, with the Mission Type set to Single Run (the default).

2 Click the Select Locale button.

The Locale list appears. Make sure the Locales folder appears in the pop-up menu.

3 For now, select the `Desert1` Locale from the list.

4 Click the Done button by the Select Locale list.

The Locale you selected now appears as the Selected Locale.

5 Click the Select IBots button.

The Select IBots list appears. The IBots folder should appear in the pop-up menu at the top of the dialog box. If it doesn't, you need to locate it.
(If you need help navigating your computer's filing system, see your operating system software manual.)

6 For now, select from the top list each of the following IBots one at a time and click the Add button: `SEEKER.BOT`, `SEEKER1.BOT`, `SEEKER2.BOT`, and `WANDERER.BOT`. Be sure to select them in the order shown.

These IBots will appear in the lower list after they are selected. To remove an IBot from the lower list, select it and click the Remove button.

7 Click the Done button to complete the IBot selection.

The IBot names you selected now appear in the Selected IBots list.

Setup Options

The Mission setup options appear below the Selected Locale. The descriptions below tell you what happens when you check these options.

- *Enable Objects* — Activates any special objects, such as goals, that may be on the Locale.
- *Random Placement* — The selected IBots are placed at random starting locations on the Locale.
- *Show Grid* — A grid of squares appears over the Locale, showing the Locale units.
- *Start Paused* — The IBots and Locale are loaded, but the Mission stays paused until you continue it. You can select an IBot to watch or debug before the action begins.
- *Test Only* — The Mission runs without keeping statistics.
- *Reset Statistics* — The previous statistics for each IBot are cleared before the Mission begins.

8 Make sure the Start Paused button is checked an then click the Run button. (Clicking the Done button will exit the Setup dialog without starting the Mission.)

After a few seconds, the Locale window and IBot status boxes display. The IBots do not move yet, because the Mission is in a paused state.

IBot Status Boxes

To the right of the Locale window, each IBot has its own Status Box that displays information showing how well the IBot is doing in the Mission. Here is a sample Status Box, with descriptions of each part:

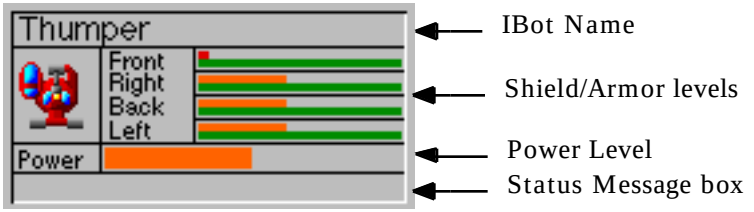


Figure 2-1: IBot Status Box

- *IBot name* Name of the IBot program.
- *IBot icon* A picture that represents the IBot on the Locale.
- *Shield/Armor levels* Bar graphs that show the current levels of shields and armor (front, right, left, and back). Each level starts as green (good condition), then changes to orange to indicate a warning level (getting low), and changes to red to indicate a danger level (low). Shield levels don't appear unless the IBot program boosts them.
- *Power level* Bar graph showing the IBot's current power level.
- *Status Message box* Area that displays any IntelliBots messages, such as errors or helpful information.

Running a Mission

When you run the Mission in step 9 below, you will see these things:

- IBots moving around on the Locale.
- Laser fire, which appears quickly as a long streak. The flash shows where the laser fire strikes.
- IBots launching shells that strike other IBots or objects, leaving craters (rubble).
- The Shield/Armor levels decreasing as each IBot takes damage from other IBots or Locale objects.
- The power levels gradually decreasing for each IBot.
- When a shell or laser hit causes an IBot's power level to reach zero, the IBot program terminates, and the IBot explodes, leaving a crater. (If the IBot's own commands cause it to run out of power, its icon remains on the Locale, but its program stops.)
- Remember that you can pause the action at any time by choosing Pause from the Mission menu.

With the above points in mind,

9 Choose Continue from the Mission menu to start the action. Let the Mission continue for about thirty seconds, then pause the action.

View and Speed Options

Before or after a Mission starts, you can change the view or speed of the action.

The following view options are available from the Views submenu under the Mission menu:

- *Auto* When a Mission starts, IntelliBots uses the Auto view mode to automatically select the view size.
- *200%*, *100%*, *75%*, *50%*, or *25%*

200% magnifies the actual Locale size by twice; 100% is the actual Locale size; 75%, 50%, and 25% reduce the view to 75%, 50%, or 25% of the actual Locale size.

- *Display Off* The Locale goes blank so the Mission can run at its fastest pace, and the IBot Status Boxes reflect the fast action.

Important: Once you choose Display Off, the View and Speed submenus are disabled until the Mission ends.

Here are some ways to change the view of the Locale:

10 Choose 200% from the Views submenu of the Mission menu.

More detail now appears in the Locale, but the edges of the `Desert1` Locale are now off the display.

11 Try these other views: 100%, 75%, 50%, and 25%.

12 Choose Continue from the Mission menu and run the Mission to its end.

Dragging the Locale

If the current Locale doesn't fit entirely on the screen, the mouse pointer will appear as a hand instead of a crosshair. You can then click on the Locale and drag it to see other parts. The Locale scrolls to let you see other parts that previously weren't displayed.

Changing the Speed

You can also change the action speed of a Mission by choosing Fast, Medium, or Slow in the Speed submenu of the Mission menu. These options affect how soon the Mission ends, but don't affect the current view.

Running Competitions

To get a better idea of how well an IBot competes, you can set the Mission Type to 3 Runs or 5 Runs from the Setup dialog. Better still, you can use one of the Competition modes that let IBots compete one-on-one in multiple Missions. The three Competition modes are Challenge, Round

Robin, and Race. Because a competition might take quite a while to finish, IntelliBots uses the Display Off view mode to speed up the action.

Challenge

In a Challenge, one IBot competes against one or more challengers. When you set up a Challenge, the first IBot you select is the *primary* IBot and the remaining IBots are its *challengers*. The primary IBot competes against each challenger, one at a time.

Each challenger is ranked by how well it performs against the first IBot. This is a good way to compare different IBots, because they are all tested against the same IBot.

Race

In a Race, each IBot competes alone against the clock. The IBots are ranked as to how long they took to accomplish a task, such as reaching a goal on a Locale.

Round Robin

In a Round Robin, each IBot competes with every other selected IBot, one-on-one. For example, if IBots A, B, C, D, and E are selected, there would be 10 Missions with these match-ups:

1) A and B; 2) A and C; 3) A and D; 4) A and E; 5) B and C;
6) B and D; 7) B and E; 8) C and D; 9) C and E; 10) D and E.

At the end, the IBots are ranked by how well they did in all the Missions of the Round Robin.

To run a Round Robin,

1. Choose Setup from the Mission menu.
2. Set Mission Type to Round Robin.
3. Select the IBots and Locale to use, as explained in *Setting Up a Mission* at the beginning of this chapter.

   For now, use the `Seeker1`, `Seeker2`, and `Wanderer` IBots and the `Desert1` Locale.
4. Choose Continue from the Mission menu to continue the action.

Because the Display Off option is used for a Round Robin, the Locale goes blank, but the Status Boxes show the continuing IBot statuses.

5 Let the Round Robin continue until its Missions end. The Mission Results dialog will then display the statistics for the Round Robin.

IBot Statistics

You can get statistics about how well IBots did in a Mission in any of these ways: the Results dialog, Get IBot Info, and Print Report.

Results Dialog

When a Mission ends, the Mission Results dialog shows the rank of each IBot for the Mission and the last message displayed for each IBot. The last message tells how the IBot ended the Mission. A sample Mission Results dialog is shown below.

Figure 2-2: IBot Statistics

Get IBot Info

To get more detailed statistics for an IBot,

1 Choose Get IBot Info from the File menu.

2 Select the assembled (.BOT) file for the IBot you want to examine.

A dialog appears showing the statistics kept for each IBot and the scores which form the final rank.

You can set the statistics to their initial (zero) values by clicking on the Reset button at the bottom of the dialog. Once this has been done, the original statistics cannot be recovered. Statistics are also reset when you build the IBot program or when the IBot participates in a Competition. In all other cases statistics are accumulated (built-on).

Below is a sample Get IBot Info dialog with explanations for each item. In each case, a higher score means better performance.

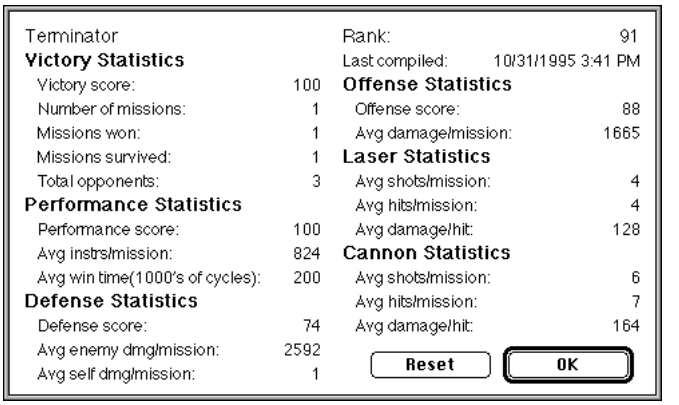


Figure 2-3: IBot Information

Statistic Type	Description
Rank	Combination of Victory Score, Performance Score, Defense Score, and Offense Score
Victory Statistics	
Victory score	Success rate
Number of missions	Total number of missions (parts of a Competition) run by this IBot
Missions won	Total number of victories for this IBot
Missions survived	Total number of missions where IBot was not destroyed
Total opponents	Number of opponents this IBot competed against
Performance Statistics	
Performance score	How well the IBot performed its designated task.
Avg. instrs/mission	Average number of CPU instructions run by IBot per mission

Avg. win time	Average time needed for each of this IBot's victories, in timer units
Defense Statistics	
Defense score	How well IBot avoided damage
Avg. enemy dmg/mission	Average units of damage caused by all opponents to this IBot, per mission.
Avg. self dmg/mission	Average units of damage this IBot caused itself, per Mission
Offense Statistics	
Offense score	How well IBot detected and damaged opponents
Avg. damage/mission	Average units of damage this IBot caused to all opponents, per mission
Laser/Cannon Statistics	
Avg. shots/mission	Average number of times laser/cannon fired per Mission
Avg. hits/mission	Average times laser/cannon hit opponent, per Mission
Avg. dmg/hit	Average units of damage this IBot caused to opponents per laser/shell hit

Print Report

For Competitions only, you can get brief printed statistics from the Print Report feature. Here's how to use the Print Report feature:

1. After running a Competition, choose Print Report from the File menu.
2. Select the report you want to print. The report title contains the date and time you generated the report.
3. Select any printing options you want from the Print dialog that appears. Check that your printer is on and set up correctly, then click OK to print the report.

Print Report displays the event type, date of event, total time of event, the Locale, and the ranking of IBot programs in the event. Here's a sample report:

Event Type:	Challenge Competition
Date of Event:	5/31/1995 6:19 PM
Elapsed Time for Event:	2 Minutes
Locale Used for Event:	Arena

1.	Sitting Duck	92
2.	Seeker	41
3.	Blocker	9
4.	Move Slow	0

Stopping the Action

To stop a Mission with no winner, choose Terminate from the Mission menu, then click the Terminate button.

Important: When you stop a Mission, all statistics for the Mission are discarded.

Summary

Concepts

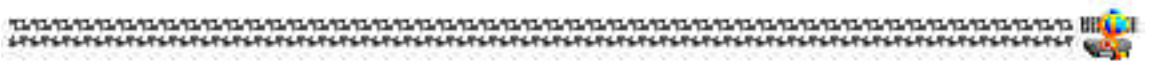
These concepts were discussed in chapter 2: *Running IBot Programs*:

A) The Setup feature selects IBots and Locales for a Mission.
B) Run (from the Mission menu or Setup dialog) runs a Mission using the IBots and Locale you selected.
C) View options are *Auto*, *200%*, *100%*, *50%*, *25%*, and *Display Off*.
D) Speed options for Missions are *Fast*, *Medium*, and *Slow*.
E) If a Locale is too large to fit in the Locale window, you can drag it to display any part of the Locale.
F) IBot status boxes show the armor/shield levels, power levels, and status messages during a Mission.
G) At the end of the Mission, the Mission Results dialog shows each IBot icon, the IBot names and current ranks, and the last messages displayed for each IBot.
H) Get IBot Info shows detailed statistics for IBots in Missions.
I) The Challenge feature matches an IBot against one or more challengers, one-on-one.
J) In a Race, each IBot competes alone against the clock.
K) In a Round Robin, each IBot competes against every other IBot, one-on-one.

Terms to Know

After reading this chapter, you should be able to briefly define the terms below. If you need help, reread the chapter or see *Glossary* in this manual.

1) Mission Setup; 2) Random Placement; 3) IBot Status Box; 4) object;
5) Speed options; 6) View options; 7) Display Off; 8) Challenge;
9) Competition; 10) Round Robin; 11) Race; 12) Results Dialog;
13) Get IBot Info; 14) Print Report



Missions

MISSION 2.1 Run two Missions on the two Locales listed below using any IBots. You can keep the action paused. Then identify and write down the objects on each map that cause heavy damage or power damage to IBots.

- Locale 1: `Arena`
- Locale 2: `Desert2`

MISSION 2.2 Run a Challenge with five IBots on the `Desert1` Locale. Decide which IBot is the best and why.

3 - Changing and Assembling Programs

In this chapter you'll learn how to change and assemble IBot programs. Here are the main topics in this chapter:

- About Computer Languages
- Looking at Source Code
- Comments, Labels, and Instruction Lines
- Changing Source Code
- Assembling Source Code
- Directives

About Computer Languages

There are three types of computer languages used today: *machine languages*, *assembly languages*, and *high-level languages*. Assembly and high-level languages are written with *source code* instructions that appear more or less like English-language statements. The source code instructions are then translated by an assembler or compiler into *machine language* that the computer can run.

MACHINE LANGUAGE Machine language is the basic set of instructions interpreted by a computer. It's usually very difficult to look at actual machine language code and understand it, so other programming languages such as assembly and high-level languages were developed, to represent machine language in a more understandable way.

ASSEMBLY LANGUAGE Assembly language was created to help programmers write easier instructions that could be converted to machine language. Learning assembly language is a good way to understand the basics of how a computer works internally. This can also prepare you to learn high-level languages, such as C and Pascal.

Each kind of assembly language is directly related to the machine language of a specific type of computer. For example, assembly for the Macintosh is

somewhat different from assembly for the PC. So, assembly programs are not generally "portable" between computer types.

High-level Languages

High-level languages were created to help simplify the task of using computer languages. While this can make programming more convenient, it doesn't help you understand the internal workings of the computer as well as assembly language does.

Usually, each high-level language instruction translates into *several* machine-language instructions. Because high-level languages are not directly related to a computer's machine language, high-level programs can run more easily on many different types of computers, so these programs are more portable between computer types.

IntelliBots Programming Language

IntelliBots uses an assembly language that contains instructions from several of today's popular assembly languages. Learning IntelliBots assembly helps you master the assembly languages of today's computers.

Assemblers

An *assembler* is a program that translates assembly code into machine language. The assembler checks the source code to make sure it's written correctly. It displays warnings or error messages for problems or errors it finds in the source code. If it does not find any source code errors, the assembler *assembles*, or converts, the source code into machine code.

Compilers

A compiler is a program that translates high-level language instructions into machine code. The compiler examines the source code and makes sure it's written correctly. It also produces warnings or error messages for problems or errors it finds in the source code and *compiles* (converts) the source code into machine code.

Looking at Source Code

A source code file contains the English-like instructions that are used to create a computer program. To open a source code file,

1 Choose Open from the File menu.

A list of source code files appears. The names of these source code files all end in ".ASM" for convenience.

2 For now, select the GOAL.ASM file.

The Edit Window appears, displaying the source code for the GOAL.ASM program:

```
Find_Goal
    COPR    #ChassisMove,#3          ;move IBot 3 units
    COPR    #ChassisTurn,#90         ;turn IBot chassis 90 degrees - turn right
    COPR    #ChassisMove,#3          ;move IBot 3 units
    COPR    #ChassisTurn,#270        ;turn IBot chassis 270 degrees - turn left
    COPR    #ChassisMove,#15         ;move IBot 15 units
    COPR    #ChassisTurn,#90         ;turn IBot chassis 90 degrees - turn right
    COPR    #ChassisMove,#4          ;move IBot 4 units
    HALT                             ;stop the IBot program
```

The elements of these source code lines are explained in *Comments, Labels, and Instruction Lines* below. (For information on degrees and directions for turning, see *Using Degrees* after the *Summary* section of this chapter.)

When assembled, this simple program moves an IBot four times and turns it three times. If this program runs alone on the Arena Locale, the next-to-last line of the program (`COPR  #ChassisMove,#4  ;move IBot 4 units`) takes the IBot to the goal on the Locale, winning the Mission. The path taken by the IBot to the goal is shown in the diagram below.

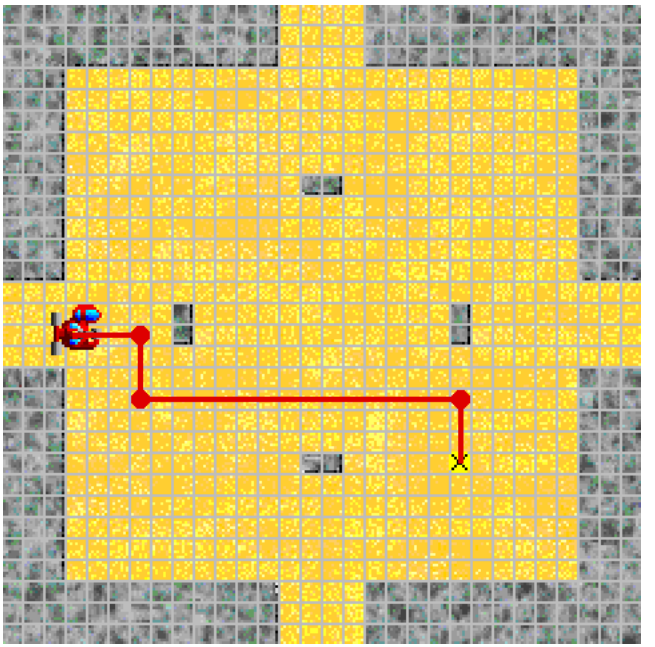


Figure 3-1: Path to goal on Arena Locale

Program Editor

The Program Editor lets you type instructions to create and change programs. It works like a basic word processor, with standard editing and filing features such as these:

- Open, close, and save files
- Positioning controls, such as: up, down, left, right, Home, End, Go To, beginning and end of line
- Text search and replace
- Cut, copy and paste
- Font and size selection
- Auto-indent and tabbing
- Automatic highlighting for program comments
- Line numbering

The default font is mono-spaced, meaning that all its characters are evenly spaced. This makes it easier to align numbers and letters vertically. You can also use a proportional font (uneven character spacing) but this makes character alignment more difficult.

Comments, Labels, and Instruction Lines

The basic elements of source code are *comments*, *labels* and *instruction lines*.

Comments

The GOAL.ASM program contains *instruction comments* to give more details about the program's instructions. In the Edit Window, these comments appear in red on a color monitor. Good, clear comments are very important to you the programmer and anyone else reading your program.

Labels

This program also contains a *label* (Find_Goal). A label acts as a reference marker in the source code, like a bookmark. It also describes the purpose of the instruction lines that follow it. When a label is created, it *must be typed at the left margin*. If you indent the label, an error occurs when you try to assemble the program.

If you create a label in a program, but you never use it in the program, the assembler will display a warning message. When you assemble the GOAL.ASM program, you will see the warning. For now, you don't need to worry about this message. You will learn more about labels and their use in *Chapter 5: Loops*.

Instruction Lines

The GOAL.ASM program has several *instruction lines*, such as the one below:

```
COPR    #ChassisMove,#3        ;move IBot 3 units
```

The basic form of an instruction line is:

```
opcode  <operand1>,<operand2>  ;comment
```

Each instruction line has these elements:

- One or more Tabs at the left margin.
- An *opcode* (such as COPR), that tells the computer the name of the instruction to *execute*, or carry out.
- At least one Tab after the opcode.
- From zero to two *operands* (depending on the opcode, and separated by commas) that tell the computer *how* to execute the instruction. In the instruction above, #ChassisMove and #3 are the operands.

Important: If an operand begins with a pound sign (#), it is called a *constant*. If you leave out the pound sign for a constant, program or syntax errors will result.

- An instruction comment that tells you what the instruction line does. Each comment *must* begin with a semicolon so the assembler won't mistake the comment for another instruction. If a comment is longer than one line, the text on the next line must also begin with a semicolon.

  Comments are very helpful for explaining what the source code does. However, when a file is assembled the comments stay in the source code; they aren't included in the resulting machine code.

Changing Source Code

You can change IBot source code instructions to make a program work differently. However, your changes must follow the IntelliBots rules for *syntax*, or grammar for source code, so the instructions will be assembled correctly. Syntax rules also apply to just about any other kind of programming language.

PRACTICE EXERCISE Suppose you want to make the IBot move around the outer edge of the `Arena` Locale to the goal, as in the example below.

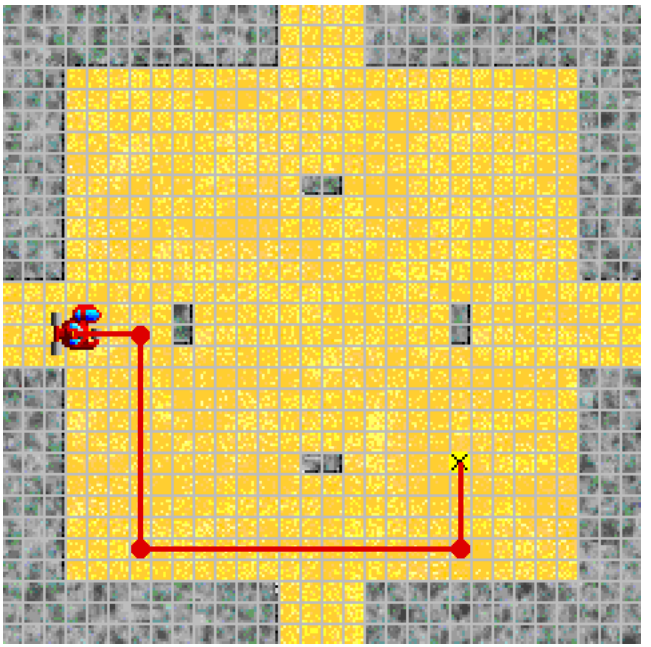


Figure 3-2: New path to goal on `Arena` Locale

To do this, you need to change the third and sixth instructions:

1. Change the third instruction (`COPR #ChassisMove,#3 ;move IBot 3 units`) so it looks like this:

   ```
   COPR    #ChassisMove,#10       ;move IBot 10 units
   ```

   All you need to do is change each 3 to a 10, and make sure the first 10 ends up as `#10` (with the pound sign). If you need help with basic text editing, see your manual for system software or word processing.

2. Change the sixth instruction (`COPR  #ChassisTurn,#90  ;turn IBot 90 degrees - turn right`) so it looks like this:

   ```
   COPR    #ChassisTurn,#270      ;turn IBot 270 degrees - turn left
   ```

The program now looks like this:

```
Find_Goal
    COPR    #ChassisMove,#3         ;move IBot 3 units
    COPR    #ChassisTurn,#90        ;turn IBot chassis 90 degrees - turn right
    COPR    #ChassisMove,#10        ;move IBot 10 units
    COPR    #ChassisTurn,#270       ;turn IBot chassis 270 degrees - turn left
    COPR    #ChassisMove,#15        ;move IBot 15 units
    COPR    #ChassisTurn,#270       ;turn IBot chassis 270 degrees - turn left
    COPR    #ChassisMove,#4         ;move IBot 4 units
    HALT                            ;stop the IBot program
```

Using Comments

Accurate and clear comments are absolutely vital to good programs. They help you plan out and clarify your instructions, and they help other programmers understand how your program works. Below are guidelines for using comments.

- Indent your comments consistently, so they line up vertically.
- Be brief.
- Describe what the instruction is doing (and why it is doing it, if necessary). A comment should not be just a copy of its instruction.

Important: When you change a line of source code, you usually need to change the comment that goes with it.

Saving Changes

You have now edited the program, using comments to document the changes in the source code. To save your changes to the program,

1. Choose Save As from the File menu.
2. Type `CH03GOAL.ASM` as the new filename.
3. Select the IBots folder from the pop-up list to save the IBot in the correct location.
4. Click Save to save the new file.

Important: After a file has been saved once, you can use Save (not Save As) from the File menu to quickly save additional changes to the file.

"Commenting Out" Instruction Lines

Instead of deleting an instruction line you don't want to keep, you can "comment it out", or add a semicolon at the beginning of the line. That turns the entire instruction into a comment so it won't be used when the program is assembled and runs.

For example, to stop the IBot before it reaches the goal in the CH03GOAL.ASM program, you could comment out the next-to-last line so it looks like this:

```
;   COPR    #ChassisMove,#4        ;move IBot 4 units
```

and assemble and run the program. To restore the instruction, you simply remove the semicolon and assemble the program.

Search and Replace

You can use the options in the Search menu to find and change specified text in your program, whether or not that text is currently on the screen. Searches are done to the end of the program; no wrapping occurs. Type the text you want to find in the upper box. You can replace the found text in the program with "replace" text you enter in the second box.

Assembling Source Code

Once your source code is complete, you need to assemble the file to translate it into machine code. To assemble the CH03GOAL.ASM file,

1 Choose Build "CH03GOAL.ASM" from the File menu.

or

If you have closed the CH03GOAL.ASM program, choose Build from the File menu, then select CH03GOAL.ASM and click OK.

The following things now happen:

A) The IntelliBots assembler checks your source code for proper syntax.
B) If the syntax is correct, the source code is translated into machine code. This machine code is saved in a new file with an extension

(ending) of ".BOT". The message "Assembly Completed Successfully" appears.

C) If the syntax has one or more errors, the message "Assembly Terminated. Errors Prevented Successful Assembly" appears with one or more error messages, and the source code is not translated.

One or more warning messages may also appear. Warning messages do not prevent the assembly from completing, but they indicate that the program *might* have a problem when it runs.

Important: IntelliBots must be able to find the `IBSYSTEM.INC` file to assemble most files correctly. Do not move or delete this file.

2 Check the messages that appear in the assembly window.

If one or more assembly errors occurred, you need to follow the steps in *Correcting Source Code Errors* later in this chapter.

Important: Renaming a .BOT file does *not* change the machine code inside it. The .BOT file is only changed by reassembling its source code program. Also, you can't edit an assembled (.BOT) file; you can only edit its source code (.ASM) file.

Running the Edited Program

To run your changed version of the program,

1 Choose Setup from the Mission menu.

2 Click the Choose Locale button and select the `Arena` Locale from the Locales folder.

3 Click the Choose IBots button and select the `CH03GOAL.BOT` program from the IBots folder.

If other filenames appear in the Selected IBots list, remove them now.

4 Check the Enable Objects and Start Paused boxes.

5 Click Run.

6 After the Locale appears, choose Continue from the Mission menu to continue the action.

The IBot will move down the Locale, turn, and move to the goal.

Correcting Source Code Errors

The most common typing errors and their assembly error messages are listed below.

Assembly Error Message	Typing Error
`Unknown opcode.`	Not using a Tab to start an instruction, or misspelling one of the `COPR` opcodes.
`Symbol(s) not defined.`	Misspelling an instruction operand.
`Syntax error.`	Forgetting the semicolon at the start of a comment, or using periods instead of commas.

Leaving out a necessary pound sign does not cause an assembly error, but usually causes problems when your program runs.

If you need to correct errors in your source code, follow the steps below; otherwise, go on to *Directives* below.

1 In the Assembly Status window, carefully note which lines in your source code are incorrect and why; then close the Assembly Status window.

If you did not close the source code file, the cursor returns to the Edit window. Otherwise, open the file as described earlier.

2 Compare your source code to the example shown in *Changing the Source Code* earlier in this lesson.

3 Correct the source code line(s) in error.

4 Choose Build "`CH03GOAL.ASM`" from the File menu.

Directives

A directive is a special instruction that tells the assembler to do a certain task *during* the assembly of the source code. Most directives are used only as instructions for the assembler, so they don't show up in the assembly output. Two of the IntelliBots directives are described below; others are described in later chapters.

.NAME <IBotName>

`.NAME` lets you select an internal name for your IBot program. This name displays in the status box during a Mission and in the title bar of some IntelliBots dialog boxes. It does not replace the program's filename.

The *IBotName* must be enclosed in quotes, but you can't include quotes inside the name itself. It can be any length, but only about 16 characters will be displayed in the IBot status box (depending on the font you use).

Suppose the filename of your program is FindGoal, but you want to display *Dr. Tracker* while IntelliBots is running. Use this `.NAME` directive:

```
.NAME   "Dr. Tracker"
```

.SUPPRESS

`.SUPPRESS` lets you stop the display of assembly warning messages. This can make assembly error messages easier to spot in the assembly window.

Summary

Concepts

These concepts were discussed in chapter 3: *Changing and Assembling Programs*:

A) The basic kinds of programming languages are machine, assembly, and high-level languages.
B) The Edit Window is a basic text processor for displaying and editing source code.
C) The Open command retrieves a source code file into an edit window.
D) A label marks a location in the code for reference. Labels usually describe instructions that follow; they must begin at the left margin.
E) An instruction line contains an opcode, from zero to two operands, and a comment. Instruction lines must begin with at least one Tab.
F) Comments are notes that explain the source code instructions. They must always begin with a semicolon.
G) The Build command assembles the source code file in the active Edit Window or from a chosen file.
H) The assembler checks your source code for correct syntax. The assembler either displays error messages, or translates the file into machine language and saves it in an assembled (executable) file.
I) You can correct errors in your program by editing the source code file and reassembling it.
J) Assembled source code files can be run as IBot programs in a Mission or Competition.

Terms to Know

After reading this chapter, you should be able to briefly define the terms below. If you need help, reread the chapter or see *Glossary* in this manual.

1) machine language; 2) assembly language; 3) high-level language;
4) source code; 5) Program Editor; 6) comment; 7) label;
8) instruction line; 9) opcode; 10) operand; 11) syntax;
12) warning message; 13) assembly error; 14) directive

Using Degrees

IntelliBots uses the concept of *compass degrees* and *relative degrees* for turning the IBot chassis or turret. Using degrees, you can turn an IBot or its turret in any direction.

The Basics

The circumference (outer edge) of a circle is divided into 360 parts, called degrees. Using compass degrees, moving clockwise on the circle, every 90 degrees represents a new basic compass direction. Zero degrees is north (straight up on a Locale); 90 degrees is east (right on a Locale); 180 degrees is south (down on a Locale); and 270 degrees is west (left on a Locale). A basic diagram of degrees on a circle is shown below.

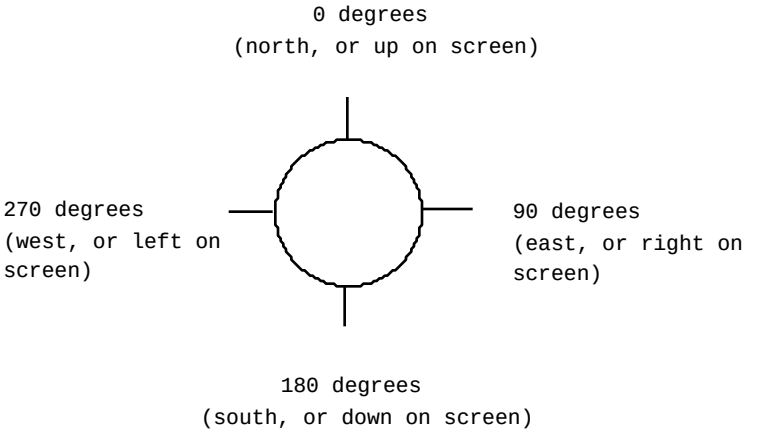


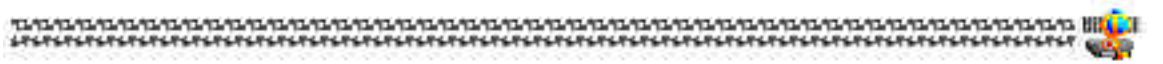
Figure 3-3 - Compass Degrees in a Circle

Using relative degrees and moving clockwise on the circle, every 90 degrees represents a new basic relative direction. Zero degrees is forward; 90 degrees is right; 180 degrees is backward; and 270 degrees is left.

Negative Degrees and 360+ Degrees

Instead of turning 270 degrees (three-quarters of a full turn) to face left of the current position, it's faster to simply turn 90 degrees to the left. This is done by using "negative degrees." For example, `COPR #ChassisTurn,#-90` turns the IBot minus 90 degrees (90 degrees to the left, or counterclockwise). You always turn to the left using negative degree values.

You can also turn an IBot or turret more than 360 degrees, which spins more than once around in a circle. For example, `COPR #ChassisTurn,#720` turns the IBot twice around the circle (360 x 2 = 720), while `COPR #ChassisTurn,#540` turns the IBot one and a half times (360 + 180).



Missions

In each Mission below, you will change the CH03GOAL.ASM program and reassemble the program to make the IBot move differently on the Arena Locale.

The source code for the CH03GOAL.ASM program is shown below.

```
Find_Goal
    COPR    #ChassisMove,#3          ;move IBot 3 units
    COPR    #ChassisTurn,#90         ;turn IBot chassis 90 degrees - turn right
    COPR    #ChassisMove,#10         ;move IBot 10 units
    COPR    #ChassisTurn,#270        ;turn IBot chassis 270 degrees - turn left
    COPR    #ChassisMove,#15         ;move IBot 15 units
    COPR    #ChassisTurn,#270        ;turn IBot chassis 270 degrees - turn left
    COPR    #ChassisMove,#4          ;move IBot 4 units
HALT                                 ;stop the IBot program
```

MISSION 3.1 Change one of the COPR #ChassisTurn,#270 instructions to look like this:

```
    COPR    #ChassisTurn,#990        ;spin the IBot
```

This makes the IBot spin around two times at the lower part of the Locale (990 degrees equals two 360-degree turns plus a 270-degree turn). Change the comments for the line above, save the program as CH03M1.ASM assemble the program, and run the Mission.

MISSION 3.2 Move the IBot along the bottom of the Locale, then up to the goal. Do not spin the IBot, as in Mission 1; instead, turn the IBot to the left when it reaches the lower right corner. Hint: To turn the IBot left, put a minus sign after the pound sign and before the 90 (such as #-90) to indicate negative 90 degrees, or 90 degrees to the left. Assemble the program and run the Mission again, saving the IBot as CH03M2.ASM.

MISSION 3.3 Using the comments in the program below, fill in the missing instructions. Then assemble the program and run it on the Arena Locale.

```
?                       ;move IBot 3 units
?                       ;turn IBot chassis 270 degrees
?                       ;move IBot 8 units
?                       ;turn IBot 90 degrees
?                       ;move IBot 15 units
?                       ;turn IBot 90 degrees
?                       ;move IBot 14 units
?                       ;turn IBot 270 degrees
?                       ;move IBot 1 unit
HALT                    ;stop the IBot program
```

Creating Programs

In this section you'll learn how to design and create programs that accomplish tasks, and how to understand data that is processed by the computer.

- *Chapter 4 - Building a New Program* explains the basic steps of how to design and create a new program.
- *Chapter 5 - Loops* shows how to write programs that repeat tasks, either conditionally or unconditionally.
- *Chapter 6 - Coprocessors and Ports* helps you understand and use IntelliBots coprocessors and ports in your program.

4 - Building a New Program

In this chapter you'll learn how to create new IBot programs of your own. Here are the main topics in this chapter:

- Program Design
- Deciding the Objective
- Dividing the Problem into Tasks
- Improving the Tasks
- Writing the Tasks as Source Code
- Implementation (Trying the Solution)
- Changing the Objective
- Introduction to the Debugger

Program Design

When you design a program, you decide the program's *objective* (what the program should do) and then divide that objective into *tasks*. Actually, design is important in almost anything you create. For example, building a model is much easier if you have a good blueprint or design to work from. The better and more clear your design is, the more successful your creation is likely to be.

Inexperienced programmers often mistakenly think it's faster and easier to finish a program by skipping the design part and typing the source code right away. But that only makes the coding time last much longer, because they end up rewriting the source code over and over to fix problems. So it's actually quicker and easier to do a good program design first; then the coding usually goes faster, and a better program is created.

Design Steps

You should follow these basic steps in program design *before* you start typing the actual instructions for your program:

Step 1: *Decide the objective for the program.*
Step 2: *Divide the objective into tasks.*

Step 3: *Improve the tasks, making them more clear and precise.*

Implementation Steps

Once you have completed these steps, you can *implement* your design (put it into action):

Step 4: *Write each task as source code.*
Step 5: *Assemble the source code and run the program.*
Step 6: *Check to see that the program accomplishes the objective correctly.*

The Cost of Errors

The earlier you catch errors, they less trouble they cost you. For example, an error in step 1 means your objective is incorrect. If you have gone all the way to step 6 before you realize the problem, you may end up rewriting the whole program. Eliminate as many errors as possible in the design phase so they won't be carried into the implementation phase.

Deciding the Objective

Step 1: *Decide the objective for your program.*

The program objective is the overall goal you want the program to accomplish. You can usually write the objective in one or two sentences. The objective should tell you *what* must be done and should tell you any important *restrictions;* but it should not tell you exactly how to solve the problem.

Restrictions

The objective below is a good one, except that it has *no restrictions*, and that opens up too many questions.

Objective 1 (weak, because of no restrictions):
Move the IBot to make it shoot the goal on the Locale.

With no restrictions, this objective raises questions like these:

- Which Locale is being used?
- Where and how far away is the goal?
- Can the IBot move anywhere, using plenty of time?
- Where is the IBot at the start of the Mission?

Here is the same objective with restrictions added:

Objective 1 (revised):

Move the IBot to make it shoot the goal on the Locale.

Restrictions:

- Use the Arena Locale.
- Move only vertically or horizontally, using the shortest path to the goal.
- Go below the south barrier.
- The goal is 18 units to the right and 8 units below the IBot's starting position.
- When you are next to the goal, shoot it with the laser.
- Your IBot is the only one in the Mission, and it starts in the *west* chamber of the Arena Locale, facing *east*.

Note: On the computer screen, north is up, south is down, west is left, and east is right.

Below is a diagram of how the IBot needs to move to reach the goal.

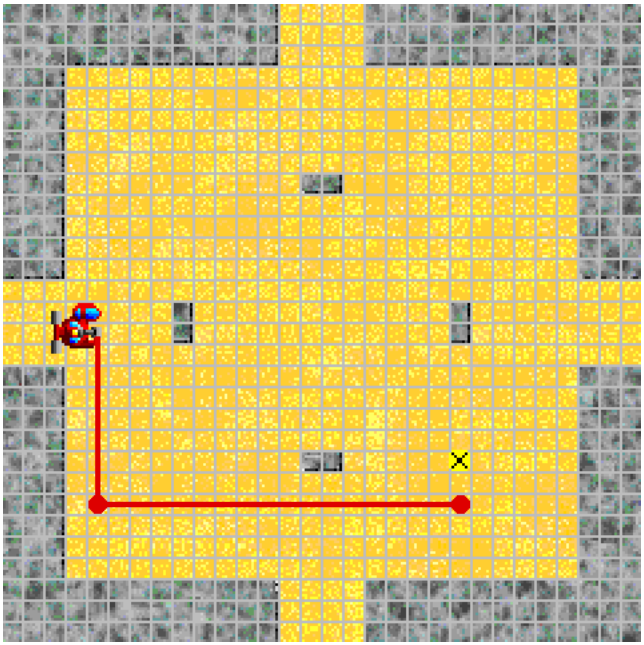


Figure 4-1: Path to goal on Arena Locale

Dividing the Objective into Tasks

Step 2: *Divide the objective into tasks.*

Looking at the previous diagram, you can divide the new Objective 1 into these basic tasks:

Task A:	Move the IBot out of the chamber.
Task B:	Move the IBot down below the south barrier.
Task C:	Move the IBot east, next to the goal.
Task D:	Shoot the goal with the laser.

Still, these basic tasks need to be improved. They don't handle the proper *turning* of the IBot for each movement; they don't tell how *far* to move; and there's no step for turning the turret to fire the laser at the correct angle.

Improving the Tasks

Step 3: *Improve the tasks, making them more clear and precise*

Task A:	Move the IBot 1 unit to the right.
Task A1:	Turn the IBot to the right (facing south).
Task B:	Move the IBot 8 units.
Task B1:	Turn the IBot right (facing east).
Task C:	Move the IBot 17 units.
Task D1:	Turn the turret to the left (pointing north).
Task D2:	Fire the laser.

The tasks now move and turn the IBot properly. The next step is to translate these tasks into the actual source code for your program.

Writing the Tasks as Source Code

Step 4: *Write the tasks as source code*

A good way to translate your tasks into source code is to type each task as a program *comment*, then fill in each instruction line with the proper instruction.

1 Choose New from the File menu.

An untitled Edit Window appears.

2 Press Tab, then type the first task as the comment for your first instruction, as shown below.

```
;move IBot right (1 unit)
```

3 Like in step 2, type the comments for each of the other tasks:

```
;turn IBot 90 degrees to face south
;move IBot (8 units)
;turn IBot left (-90 degrees)
;move IBot (17 units)
;turn turret towards goal
;shoot goal with laser
```

4 After the first tab on the first line, type the instruction that moves the IBot 1 unit. Be sure to include a tab after COPR and another tab before the comment. The finished instruction line looks like this:

```
COPR    #ChassisMove,#1    ;move IBot right (1 unit)
```

This kind of instruction is like the ones explained in *Chapter 3 - Changing and Assembling Programs*. You can be somewhat flexible in how you word the comments, as long as you accurately describe the instructions.

5 Repeat step 4 for each task, adding an instruction line for each comment. Be sure to include the commas (,), pound signs (#), and

semicolons (;) in the right locations. (Spelling mistakes inside comments won't cause assembly errors.)

The finished program now looks like this:

```
COPR    #ChassisMove,#1    ;move IBot right (1 unit)
COPR    #ChassisTurn,#90   ;turn IBot 90 degrees to face south
COPR    #ChassisMove,#8    ;move IBot (8 units)
COPR    #ChassisTurn,#-90  ;turn IBot left (-90 degrees)
COPR    #ChassisMove,#17   ;move IBot (17 units)
COPR    #TurretTurn,#-90   ;turn turret towards goal
COPR    #OffenseLaser,#20  ;shoot goal with laser
HALT                       ;stop the IBot program
```

In small programs like this one, the number of tasks will usually match the number of source code instructions. In larger programs, you will need to divide some of the improved tasks into multiple instructions.

Implementation (Trying the Solution)

Step 5: *Assemble the source code and run the program.*

1 Save the file as `CH04GOAL.ASM` in the IBots folder.

2 Assemble the file by choosing Build "`CH04GOAL.ASM`" from the File menu.

If the file does not assemble correctly, you need to correct all typing errors and assemble the file again (see *Correcting Source Code Errors* in *Chapter 3 - Changing and Assembling Programs*).

Showing the Locale Grid

This Mission depends on the IBot moving a precise number of units, so you may want to show the Locale *grid*. The grid displays gray framework lines over the Locale; each square that appears corresponds to one Locale unit.

3 Choose Setup from the Mission menu.

4 Check the Show Grid box.

Running the Mission

Now you need to set up the Mission and run it.

With the Setup Dialog open,

5 Select the Arena Locale and the CH04GOAL.BOT program.

6 Click Enable Goal, then click Run.

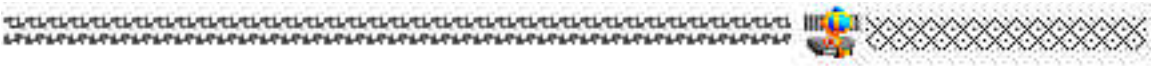
7 Select Continue from the Mission Menu.

Checking the Results

Step 6: *Check the results.*

The IBot should move to the goal and win the Mission. If the IBot does not reach the goal, correct any mistakes in your source code; then repeat steps 1 and 2 and 6 through 9 above. You will learn more about checking program results in *Chapter 12: Testing Programs*.

You have now designed and created a new program, and you have tried it out by running a Mission.



Changing the Objective

When you change a program's objective or restrictions, you usually need to redesign and rewrite the tasks. Objective 2 below is a little different from the Objective 1 you used. The changes are noted in *italics*.

Objective 2:

Move the IBot to make it *reach* the goal on the Locale.

Restrictions:

- Use the Arena Locale.
- Move only vertically or horizontally.
- Go just below the *west* barrier.
- The goal is 18 units to the right and 8 units below the IBot's starting position.
- *Run into* the goal.
- Your IBot is the only one in the Mission, and it starts in the west chamber of the Arena Locale, facing east.

Here are the improved tasks for Objective 2:

Task A:	Move the IBot 1 unit to the right.
Task A1:	Turn the IBot to the right (facing south).
Task B:	Move the IBot 2 units.
Task B1:	Turn the IBot left (facing east).
Task C:	Move the IBot 17 units.
Task D1:	Turn the IBot to the right (facing south).
Task D2:	Move the IBot 5 units.

Changing the Program

Now you need to write these tasks as source code and implement the program.

1. Choose New from the File menu.
2. Write the tasks for Objective 2 as source code instructions.

   If you need help or would like to check your instructions for accuracy, see *Source Code for Objective 2* below.
3. Save the file as `CH04GOL2.ASM`.
4. Assemble the source code without errors.
5. Choose Setup from the Mission menu.
6. Select the `Arena` Locale and the `CH04GOL2.BOT` IBot.
7. Continue the Mission action.

The IBot should reach the goal. If it doesn't, correct any mistakes in your source code and repeat steps 3 through 7 above.

Source Code for Objective 2

Here are the source code instructions for step 2 above:

```
COPR    #ChassisMove,#1      ;move IBot right (1 unit)
COPR    #ChassisTurn,#90     ;turn IBot to face down (90 degrees)
COPR    #ChassisMove,#2      ;move IBot (2 units)
COPR    #ChassisTurn,#-90    ;turn IBot left (-90 degrees)
```

```
COPR    #ChassisMove,#17     ;move IBot (17 units)
COPR    #ChassisTurn,#90     ;turn IBot right (90 degrees)
COPR    #ChassisMove,#5      ;move IBot (5 units)
HALT                         ;stop the IBot program
```

Exercise 1: Using a new objective

When you choose a new objective for a program, you need to design new tasks and write new program instructions. Here is a new objective to try:

Objective 3:

Move the IBot in a pattern so it traces a large square pattern.

Restrictions:

- Use the `Field` Locale.
- There are 36 units to each side of the square.
- The IBot turns to the right each time it completes a side.

Here are the tasks for Objective 3:

Task A:	Turn the IBot -45 degrees (minus 45 degrees, facing east).
Task A1:	Move the IBot 36 units.
Task B:	Turn the IBot 90 degrees.
Task B1:	Move the IBot 36 units.
Task C:	Turn the IBot 90 degrees.
Task C1:	Move the IBot 36 units.
Task D:	Turn the IBot 90 degrees.
Task D1:	Move the IBot 36 units.
Task E:	Halt the IBot.

1. Open a new file and write these tasks as source code.

2. Save the file as `CH04SQR.ASM`.

3. Assemble the program without errors and run a Mission with it as the only IBot on the `Field` Locale.

4. When the IBot finishes tracing the square, terminate the Mission (choose Terminate from the Mission menu).

If the program doesn't work as expected, correct any mistakes in your source code, reassemble it, and run the Mission again.

Introduction to the Debugger

About the Debugger

The IntelliBots Debugger is a full-featured debugging tool for testing programs. (*Debug* means to find and fix *bugs*, or problems in your program source code.) Instead of pausing and continuing the Mission action to figure out a problem in your program, you can get help with the Debugger.

The IntelliBots Debugger helps you step through your program one instruction at a time, so you can find problems in your source code. This can be very helpful as you write larger or more complicated programs. You will learn more about the Debugger in later chapters.

Displaying the Debugger

To display the Debugger,

1. With the action paused, run a Mission using the program you want to debug. For now, use the `CH04SQR.BOT` program and the `Field` Locale.

   Important: If you run two or more programs in a Mission, you must click the IBot's icon (in the status box) for the program you want to debug. If you run just one program, the IBot icon is already highlighted.

2. Choose Debug from the Mission menu to display the Debugger window.

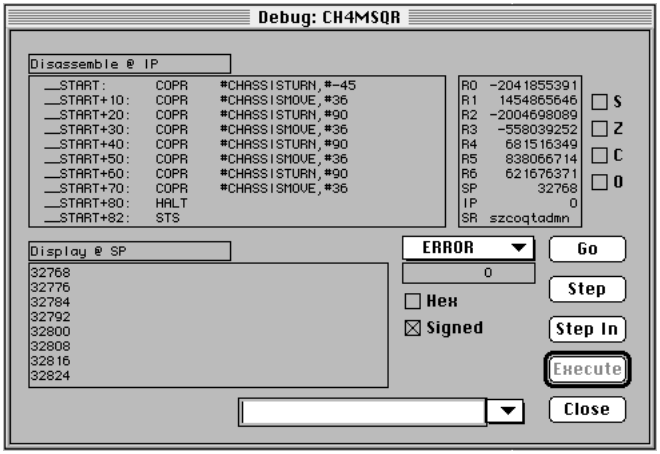


Figure 4-2: Debugger window

3 To see more of the Locale, you can drag the Debugger window to another part of the screen.

The following parts of the Debugger are discussed in this chapter:

- Disassembly window
- Step button
- Close button
- RS (ReStart) command on the Debugger command line

Disassembly Window

Disassembly means displaying machine code instructions in their original source code format, like the opposite of assembling. The right side of the Disassembly window displays the current instruction, as well as the next several program instructions. All instructions are displayed without comments. The left side of the window shows the *address* (location in computer memory) of each instruction.

For example, the first instruction in the disassembly window is `COPR #ChassisMove,#-45`. The address for this instruction is labeled as `__START`.

Step Button

When you click the Step button, you run the current instruction in the disassembly window. (If that instruction causes some kind of IBot action, you will see the action happen in the Locale window.) The next instruction then displays at the top of the disassembly window. Each time a new instruction is executed, the disassembly window scrolls. You can't scroll backwards or re-run any instruction.

1. Click the Step button once (on the right of the Debugger window).

   The IBot turns -45 degrees, and the disassembly window scrolls to the next instruction: `COPR #ChassisMove,#36`.

2. Click the Step button again.

   The IBot moves 36 units to the right. (When other programs are in the Mission, they will also execute their instructions in turn, but you will only see instructions for the program you are debugging.)

Close Button

To close the Debugger window,

1. Click the Close button on the right of the Debugger window.

   The Mission continues as before. If the Mission was in a paused state, it remains paused but shows any effects of instructions you have run in the Debugger.

RS Command

The RS (ReStart) command closes the Debugger window and restarts the current Mission. The RS command is especially useful when you have stepped too far in the program, and you need to restart the Mission.

1. Choose Debug from the Mission menu to display the Debugger window.

2. Type RS on the Command line (at the bottom of the Debugger window) and press Return.

.LOCK <FILENAME> The .LOCK assembler directive lets you lock (encrypt) your assembled file so its source code can't be seen in the Debugger. If you give an assembled file to someone else to run in a Mission, you may want to lock the file so your opponent can't use the Debugger to see the program contents. The locked file will run normally in IntelliBots. To unlock the program, remove the .LOCK directive from the source code and reassemble the file.

Summary

CONCEPTS

These concepts were discussed in chapter 4: *Building New Programs*:

A) Program design is the process of deciding the objective for your program and then dividing the objective into the correct tasks.
B) To design a program, you follow these steps:
   1) Write the objective.
   2) Divide the objective into tasks.
   3) Improve the tasks, making them more precise and clear.
C) To implement a program, you follow these steps:
   4) Write the tasks as source code.
   5) Try out the solution.
D) When an objective changes, the tasks should be changed.
E) The check box Show Grid in Mission Setup displays a grid over the Locale.
F) The Debugger is a software tool that lets you see a program's instructions being executed.
G) The Debugger is only active when a Mission window is displayed and a IBot icon is selected.
H) The disassembly window displays the program instructions; it does not display comments.
I) The Step command executes your program instructions one at a time.
J) The RS command (ReStart) closes the Debugger and restarts the current Mission.
K) The .LOCK directive prevents an assembled program from being seen in the Debugger.

TERMS TO KNOW

After reading this chapter, you should be able to briefly define the terms below. If you need help, reread the chapter or see *Glossary* in this manual.

1) program design; 2) task; 3) implementation; 4) objective; 5) restriction; 6) Debugger; 7) disassembly; 8) Step; 9) Close; 10) RS (ReStart) 11) .LOCK

Missions

The Missions below use the `Field` Locale. Design and implement the program for each exercise below. Test each program by running a Mission with your IBot on the `Field` Locale.

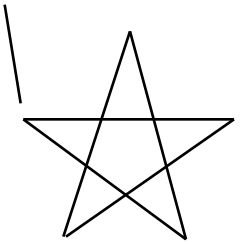
Important: There are many variations you can try on the Missions and exercises in this chapter and later chapters. Once you master them the regular way, design your own new objectives to try some variations.

MISSION 4.1 Move the IBot in a pattern so it traces a *triangle.* Each side is 40 units, and the IBot turns 120 degrees each time. Save the file as `CH04M1.ASM` in the Missions folder.

MISSION 4.2 Move the IBot in a pattern so it traces a *octagon.* The IBot starts at the upper right corner of the octagon. Each side is 14 units; each angle is 45 degrees. Save the file as `CH04M2.ASM` in the Missions folder.

MISSION 4.3 Move the IBot in a pattern so it traces five lines to form a *star.* The IBot starts at the upper left tip of the star. Each line is 30 units in length; each angle the IBot must turn is 144 degrees. Save the file as `CH04M3.ASM`.

(start)



5 - Loops

In this chapter you'll learn how to write programs that repeat tasks, either conditionally or unconditionally. Here are the main topics in this chapter:

- Infinite Loops
- Building an Infinite Loop
- Conditional Loops
- Building a Conditional Loop
- Using Variable Data and Registers
- Decrementing a Value
- Conditional Jumps
- Incrementing a Value

A *loop* is a repeating set of program instructions. An *infinite* loop repeats forever unless something outside the program interrupts it (such as turning off the computer). A *conditional* loop repeats until a certain condition is reached. A computer can easily repeat a set of instructions thousands or millions of times, but it's very difficult for us to write that many separate instructions. A loop lets the computer do the work of repetition, while we simply tell it what to repeat and when. Infinite loops and conditional loops are explained in this chapter.

Infinite Loops

An infinite loop lets you repeat a set of instructions indefinitely, until you terminate the program (such as by choosing Terminate from the Mission menu). A familiar business-world example of an infinite loop is an electronic "banner" sign that shows a moving message and repeats it when the message goes off the sign at the edge.

Sometimes a programmer will accidentally create an infinite loop where none was supposed to exist. When this happens, the program is said to be "stuck in a loop." It's important to create infinite loops correctly when they're wanted, and avoid them when they're not wanted.

Building an Infinite Loop

To see how an infinite loop works, you can change the square-tracing pattern you wrote in Chapter 4 into a loop program. That way, the IBot will keep tracing the same square shape until you stop the Mission. Here are the original tasks for that program:

Task A: Turn the IBot -45 degrees (to point east)
Task A1: Move the IBot 36 units.
Task B: Turn the IBot 90 degrees.
Task B1: Move the IBot 36 units.
Task C: Turn the IBot 90 degrees.
Task C1: Move the IBot 36 units.
Task D: Turn the IBot 90 degrees.
Task D1: Move the IBot 36 units.

To repeat this set of instructions over and over, add a task at the end telling the program to go back to Task A.

Task E: Go to Task A.

This *go to* task creates an infinite loop, because the program keeps repeating tasks A through E.

Avoiding Loop Errors

When you change a regular program into a loop, you might run into a few problems. To avoid problems with loops,

- Watch for any tasks that should *not* be repeated, and put them before or after the loop.

  In the square-tracing program, Task A should only be done *once*, so the IBot points south. If Task A gets repeated, the square will be tilted left 45 degrees each time it is traced.

  To fix this, Task E should be:

  Task E: Go back to Task A1.

Now Task A is *before* the loop, and Task A1 is the *first* task of the loop.

- Make sure there are no missing or extra tasks when the last "action" task of the loop moves to the first task of the loop.

  The last IBot movement task is Task D1: Move the IBot 36 units. When the loop goes back to A1, then Task A1 says to move 36 units, also. At that point, the IBot moves 72 units without turning, which ruins the square shape.

  To fix this, add task D2:

  Task D2: Turn the IBot 90 degrees.

- If you want the loop to run faster (and exact dimensions aren't important), make any long task into a shorter one.

  For example, the IBot could move just 10 units instead of 36 units each time.

FIXING THIS LOOP Using the three suggestions above, the tasks now look like this:

```
    Task A:    Turn the IBot -45 degrees (to point east)
Start of Loop
    Task A1:   Move the IBot 10 units.
    Task B:    Turn the IBot 90 degrees.
    Task B1:   Move the IBot 10 units.
    Task C:    Turn the IBot 90 degrees.
    Task C1:   Move the IBot 10 units.
    Task D:    Turn the IBot 90 degrees.
    Task D1:   Move the IBot 10 units.
    Task D2:   Turn the IBot 90 degrees.
    Task E:    Go back to Task A1.
```

However, you can write this loop with fewer instructions. Notice that there are really only *two different* instructions inside the loop: move 10 units, and turn 90 degrees. To save yourself some typing, you could write the loop like this:

```
Start of Loop --------
    Task A1:   Move the IBot 10 units.
```

Task B: Turn the IBot 90 degrees.
Task E: Go back to Task A1.

Now the IBot traces the complete square after the loop runs *four times*, but only three printed instructions are used in this loop instead of nine for the original loop.

The next step is to translate these tasks into source code. You need to put a *label* in the source code to indicate the start of the loop, and task E needs a special *jump* instruction to go back to the start of the loop.

LABEL AND JUMP

A *label* is a title you create to show *where* the program jumps. Label names begin with a letter and can contain letters, numbers, or underscores (_), but not spaces. So instead of "Start of Loop", we can use "Start_of_Loop" (replacing the spaces with underscores).

The JUMP (Jump) instruction makes the program go back to the label you use. So, Task E can be translated to this:

```
    JUMP    Start_of_Loop
```

The source code translation for the tasks looks like this:

```
    COPR    #ChassisTurn,#-45    ;turn -45 degrees (pointing east)
Start_of_Loop
    COPR    #ChassisMove,#10     ;move IBot 10 squares
    COPR    #ChassisTurn,#90     ;turn 90 degrees
    JUMP    Start_of_Loop        ;go back to start of loop
```

To try this new program,

1. Open a new file.

2. Type the program instructions above.

   Be sure to type Start_of_Loop at the left margin, but begin all other instructions with a Tab.

3. Save the program as CH05SQR.ASM in the IBots folder and assemble the program without errors.

4. Run a Mission on the Field Locale with the CH5SQR.ASM IBot.

5 After the IBot completes the square several times, stop the action.

Jumping Ahead

You can also use the JUMP instruction to jump ahead in the program (although jumping ahead does not usually cause a loop to happen). Suppose you want to trace *either* a square, a triangle, or a diamond. Instead of writing three separate programs, you can write one program and include a jump instruction at the right place to skip ahead to the right part of the program. That way, you don't have to keep deleting or adding instructions every time you want to change the program.

In the example below, the program skips over the square-tracing routine to do the triangle-tracing and square-tracing routines. (The dots (...) mean that you would write more tasks at those points.)

```
Start
    Jump to Trace_Triangle
Trace_Square
    ...
Trace_Triangle
    ...
Trace_Diamond
    ...
```

Conditional Loops

A conditional loop repeats a set of instructions only until a certain condition exists. A familiar example of a conditional loop is an ATM cash machine at the bank. The ATM's computer program keeps displaying a "welcome" screen until you press a button; then it displays a different screen of instructions, based on the choice you made. The ATM program uses conditional loops to guide you through its screens, until you finish with your transaction. These conditional loops are actually contained within an infinite loop, because when the transaction is done, you start all over with the first screen.

If a conditional loop runs but never finds the condition it's looking for, the loop will become infinite. Plan your conditional loops carefully so they don't turn into infinite loops by mistake.

Building a Conditional Loop

The CH05SQR.ASM program you wrote always moves the IBot 10 units. What if you wanted to move the IBot 10 units, then 9, then 8, etc., down to zero? Instead of a square, the IBot would trace a shrinking square spiral:

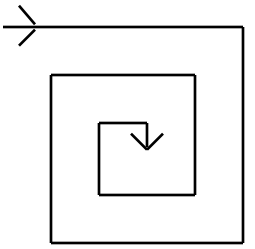


Figure 5-1: Shrinking square spiral for conditional loop

Objective: Trace a shrinking square spiral, starting at 10 units and decreasing by one on each side, until the size is zero.
Here are the tasks in planning this conditional loop:

Task A:	Turn the IBot 45 degrees to the left.
Task B:	Set the starting length (number of units) of the square side as 10.
Task C:	Begin the conditional loop with a label, such as Start_Loop.
Task D:	Move the IBot the number of units in the current length.
Task E:	Turn the IBot.
Task F:	Subtract one from the current length.
Task G:	If the length is more than zero, repeat the loop; else exit the loop.

Tasks A and B are put *before* the loop. If they were put *in* the loop, two problems would happen each time the loop repeated: the IBot would keep turning an extra 45 degrees; and the size would keep getting reset to 10 each time the loop repeated, so the size would never reach zero.

Tasks C, D, and E are like the ones used in the infinite loop discussed earlier. Tasks F and G are new; their concepts are discussed below, after tasks A and B.

Using Variable Data and Registers

Task A: *Turn the IBot 45 degrees to the left.*
Task B: *Set the starting size (number of units) of the square side as 10.*

Variable Data

The length of the side starts as 10 units, but it will gradually decrease until it becomes zero. This length is called *variable data*, because it constantly changes. So, the variable data starts as 10, then drops by one each time the loop repeats (to 9, to 8, etc.). Your program needs to *set* the value of the variable data and *store* the value where it can be easily accessed. You can set and store values in a *register*.

Registers

The computer's CPU provides useful storage locations called *registers*. Registers are named R0 (register zero), R1 (register 1), and so on, up to R6. You can think of a register as a scratch pad for writing and holding data. A register can only hold one value at a time, but you can change that value as often as you want.

MOVE

To get a value into a register, you can use the `MOVE` instruction. `MOVE` copies a value into a register or other memory location. (A register contains an unpredictable value until you move a known value into it.) In the example below, the `MOVE` instruction moves 10 into register 1 (R1):

```
MOVE    #10,R1              ;set starting length of square side
```

The square loop example uses R1, but any register from R0 to R6 could be used.

Using a Register

Now that there's a value in R1, other instructions can access that value. For example, instead of using `COPR #ChassisMove,#10`, you could use `COPR #ChassisMove,R1`. Then the IBot would move the same number of units as whatever value is currently stored in R1.

So far, the conditional loop looks like this:

```
    COPR    #ChassisTurn,#-45    ;turn IBot to face east (TASK A)
    MOVE    #10,R1               ;set starting size of side (TASK B)
Start_Loop                       ;(TASK C)
    COPR    #ChassisMove,R1      ;move IBot "R1" units (TASK D)
    COPR    #ChassisTurn,#90     ;turn 90 degrees (TASK E)
```

(F Subtract one from the current length.)

(G If the length is greater than zero, repeat the loop; else exit the loop.)

Decrementing a Value

Task F: *Subtract one from the current length.*

DEC

Now that the length is in R1, you can decrease the length by decreasing (decrementing) the value in R1. To do this, you can use the DEC instruction to decrease R1 by one:

```
    DEC     R1                   ;decrease R1 by one
```

Now the conditional loop looks like this:

```
    COPR    #ChassisTurn,#-45    ;turn IBot to face east (TASK A)
    MOVE    #10,R1               ;set starting size of side (TASK B)
Start_Loop                       ;(TASK C)
    COPR    #ChassisMove,R1      ;move IBot "R1" units (TASK D)
    COPR    #ChassisTurn,#90     ;turn 90 degrees (TASK E)
    DEC     R1                   ;decrease R1 by one (TASK F)
```

(G If the length is greater than zero, repeat the loop; else exit the loop.)

SUB

SUB (Subtract) decreases a value by the amount you choose. Instead of using DEC, you could use SUB, like this:

```
    SUB     #1,R1                ;subtract one from R1
```

If you only need to subtract one, the DEC instruction will actually run faster than SUB. For subtracting more than one at a time, SUB is the better choice. If you subtract 2 or more in the square loop, the square will shrink faster.

Conditional Jumps

Task G: *If the length is more than zero, repeat the loop; otherwise, exit the loop.*

JG

A conditional jump instruction (such as JG) causes a jump only when a certain condition occurs. For example, the CMP instruction shown sets the "zero" condition flag if the result of the instruction is zero (R1 = 0). (You will learn more about condition flags in *Chapter 8: Checking Conditions*.) If R1 is greater than zero, the JG (Jump on Greater than Zero) conditional jump instruction will repeat the loop.

```
    CMP     #0,R1               ;compare zero to R1
    JG      Start_Loop          ;if R1 > zero, go back to Start_Loop
```

As long as R1 is greater than zero, the JG will keep sending the program back to Start_Loop. But when R1 gets to zero (or below), JG no longer jumps to Start_Loop; instead, the program continues with the *next instruction* after JG. This instruction could be HALT, to stop the program.

```
    CMP     #0,R1               ;compare zero to R1
    JG      Start_Loop          ;if R1 > zero, go back to Start_Loop
    HALT                        ;stop the IBot program
```

Here's the finished loop:

```
    COPR    #ChassisTurn,#-45   ;turn IBot to face east
    MOVE    #10,R1              ;set starting length of square side
Start_Loop
    COPR    #ChassisMove,R1     ;move IBot the number of units in R1
    COPR    #ChassisTurn,#90    ;turn 90 degrees (facing right)
    DEC     R1                  ;decrease R1 by one
    CMP     #0,R1               ;compare zero to R1
    JG      Start_Loop          ;if R1 > zero, go back to Start_Loop
    HALT                        ;stop the IBot program
```

Exercise 1: Trying the Decrementing Loop

To try this decrementing loop,

1 Open a new file.

2 Carefully type the instruction lines shown above for the conditional loop.

3 Save the program as `CH05DEC.ASM` and assemble it without errors.

4 Run a Mission on the `Field` Locale with the `CH05DEC.ASM` IBot.

The IBot will trace a shrinking square spiral.

Incrementing a Value

An *incrementing* conditional loop uses an increasing value each time the loop repeats. For example, you can change the shrinking square loop into an expanding square loop by using incrementing instructions. So if the IBot starts at the center of the Locale, it could move in an expanding square spiral shape until the side length reaches 11 squares:

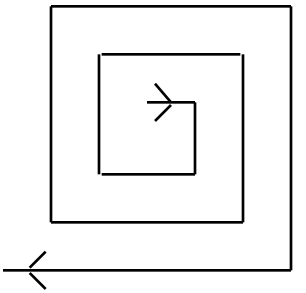


Figure 5-2: Expanding square spiral for incrementing loop

Here are the tasks for this incrementing loop:

Task A:	Set the initial side length to a small value, such as 2. Set the initial angle to point east.
Task B:	Move the IBot as much as the current side length.
Task C:	Turn the IBot to its right.
Task D:	Use the `INC` (Increment) instruction to add one to the length.
Task E:	Compare the length to a maximum limit, such as 12.
Task F:	If the length is less than 12, repeat the loop.

Here's the source code for the incrementing loop:

```
    COPR    #ChassisTurn,#-45    ;turn left 45 degrees (pointing east)
    MOVE    #2,R1                ;move starting length into R1
Loop
    COPR    #ChassisMove,R1      ;move the IBot the length of a side
    COPR    #ChassisTurn,#90     ;turn 90 degrees
    INC     R1                   ;add one to current value of R1
    CMP     #12,R1               ;compare 12 to R1
    JL      Loop                 ;if R1 < 0, repeat loop
    HALT                         ;result is 0, halt IBot
```

Notice that there are two new instructions in this program: INC and JL along with CMP which was used in the last example. These instructions are described below.

INC AND ADD

The INC (Increment) instruction increases a value by one, like the reverse of the DEC instruction. You can also use the ADD instruction to increase a value by the amount you choose, like the opposite of the SUB instruction. For adding just one, INC is a faster instruction.

CMP (COMPARE)

You can use the CMP (Compare) instruction to check the value of a register, such as R1 (where the length of the square side is being stored).

```
    CMP     #12,R1               ;compare 12 to the value in R1
```

CMP compares the two operands (12 and R1) by subtracting the first operand (12) from the second (R1). If the result is zero, the values are equal (R1 = 12). If the result is positive, then R1 is greater than 0; if the result is negative, then R1 is less than 12.

Remember: CMP expects the *first* operand to be a value (like 12) and the *second* operand to be a register (like R1).

JL

You can use the JL (Jump on Result Less Than) instruction to jump *only* if the CMP instruction results in a negative value. For example, if R1 is 11, then 11 minus 12 is negative, so the JL instruction would be executed to jump back to the Loop label. If R1 is 12, the CMP result is zero, so JL would be skipped, and HALT would be executed.

Exercise 2: Trying the Incrementing Loop

To try out the incrementing loop, type the source code for the incrementing loop in a new file named `CH05INC.ASM`. Assemble the program and run a Mission on the `Field` Locale. When the IBot completes the expanding square loop, choose Terminate from the Mission menu. If you need help with any of these steps, see the steps in *Exercise 1: Trying the Decrementing Loop* for some ideas.

Viewing Registers in the Debugger

The Registers window in the Debugger shows the current values of the seven general purpose registers, which are R0 through R6. When a program begins, R0 through R6 contain random values; then they are updated as the registers change.

To try out the Registers window,

1. With the action paused, run the `CH05INC` program on the `Field` Locale.
2. Choose Debug from the Mission menu.

   The Registers window is at the upper-right corner of the Debugger window. It currently has a random value in it.
3. Click the Step button twice, causing the `MOVE #2,R1` instruction to run.

   The next instruction is `COPR #ChassisMove,R1`. Now the R1 value displays as 2 in the window, as shown in Figure 5-3 below.

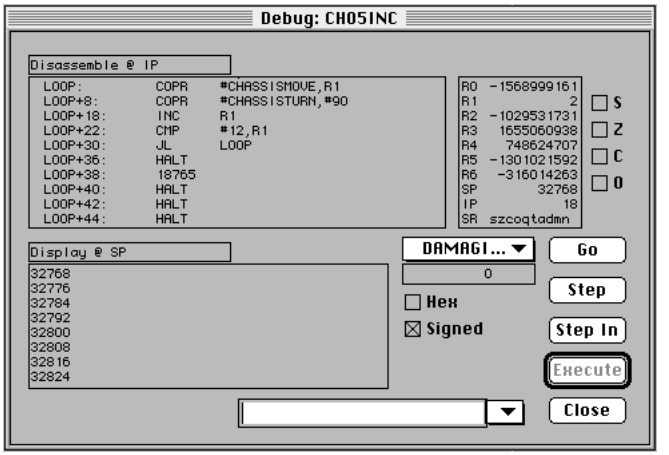


Figure 5-3: Registers window after `MOVE #2,R1` instruction executes.

4 Click the Step button three times, causing the `INC R1` instruction to run.

The next instruction is `CMP #12,R1`. Now the R1 value displays as 3 because it was incremented by 1, as shown in Figure 5-4 below.

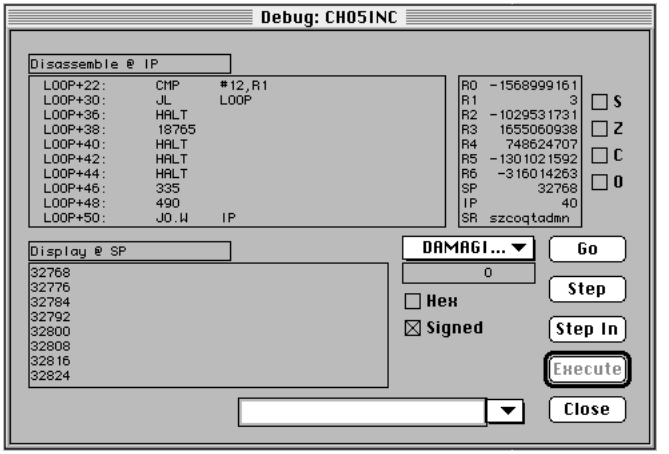


Figure 5-4: Registers window after `INC R1` instruction executes.

5 Choose Terminate from the Mission.

JE and Conditional Loops

The JE instruction jumps if the result is equal to zero. It's possible to use this instruction in conditional loops, but a serious problem may occur. If the decrementing or incrementing value ever goes *past* the value checked by the JE, the loop will continue on infinitely, which is not what you want.

In this example, the loop works because it's always incrementing by one.

```
    MOVE    #0,R1               ;move 0 into R1
Loop
    INC     R1                  ;add one to current value of R1
    CMP     #10,R1              ;compare 10 to R1 (R1 minus 10)
    JE      ExitLoop            ;if result equals 10, exit the loop
    JUMP    Loop                ;repeat the loop
ExitLoop
```

But the next example turns into an infinite loop, because the value will eventually increment from 8 to 10, passing the limit that JE checks for.

```
    MOVE    #0,R1               ;move 0 into R1
Loop
    ADD     #2,R1               ;add two to current value of R1
    CMP     #9,R1               ;compare 9 to R1 (R1 minus 9)
    JE      ExitLoop            ;if result equals 10, exit the loop
                                ;this causes an infinite loop
    JUMP    Loop                ;repeat the loop
ExitLoop
```

Therefore, it's better to use JG and JL for conditional loops. Here's the above example done correctly with a JG instruction:

```
    MOVE    #0,R1               ;move 0 into R1
Loop
    ADD     #2,R1               ;add two to current value of R1
    CMP     #9,R1               ;compare 9 to R1 (R1 minus 9)
    JG      Loop                ;if result is more than 9, exit loop
```

Summary

Concepts

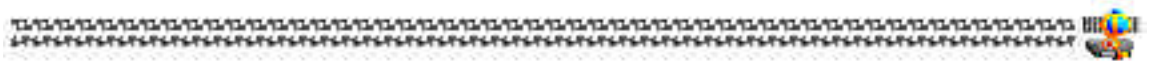
These concepts were discussed in chapter 5: *Loops*:

A) An infinite loop repeats the same instructions indefinitely, until you terminate the program.
B) A label is a location in the program used by a jump instruction.
C) `JUMP` makes the program jump to the specified label.
D) A conditional loop repeats only as long as a certain condition is met.
E) A decrementing loop uses a smaller value each time the loop repeats.
F) An incrementing loop uses a larger value each time the loop repeats.
G) A register is a location in the CPU where the program stores a value.
H) `MOVE` copies a value (like an initial value for a loop) into a register.
I) `SUB` and `DEC` decrease the current value in the register.
J) `ADD` and `INC` increase the current value in the register.
K) `CMP` compares a given value to the current value in the register.
L) A conditional jump (such as `JG` or `JL`) checks the result of a `CMP` and jumps if the result meets a certain condition.
M) When the condition for a conditional jump is not met, the program continues on to the instruction directly after the conditional jump.
N) The Registers window in the Debugger displays the current value of data items such as the general purpose registers (R0 through R6).

Terms to Know

After reading this chapter, you should be able to briefly define the terms below. If you need help, reread the chapter or see *Glossary* in this manual.

1) loop; 2) infinite loop; 3) conditional loop; 4) jump; 5) variable data
6) initialize; 7) compare; 8) move; 9) decrement;
10) increment



Missions

MISSION 5.1 Convert the triangle-tracing program you wrote in chapter 4 (`CH04M1.ASM`) into an infinite loop program. Save the new program as `CH05M1.ASM`. Assemble it and run it on the `Field` Locale. Then use the Debugger to step through the program. Watch the Registers window data as it changes.

MISSION 5.2 Create a new program that moves an IBot back and forth on a horizontal line. Save the program as `CH05M2.ASM` and run it on the `Field` Locale. Here are the basic tasks:

1 Turn the IBot to face east.
2 Make the IBot move a certain distance you choose (such as 20 units).
3 Turn the IBot 180 degrees.
4 Loop back to task 2.

MISSION 5.3 Change the program you wrote for Mission 5.2 so it moves the IBot one unit *less* each time (such as 10 to the right, 9 to the left, 8 to the right, etc.). Save the program as `CH05M3.ASM` and assemble it and run it on the `Field` Locale. Here are the basic tasks:

1 Turn the IBot to face east.
1a Put the initial distance to move (such as 10 units) into a register.
2 Make the IBot move a certain distance you choose.
3 Turn the IBot 180 degrees.
4 Add the rest of the steps you need for the program. If you need help, see the `CH05DEC.ASM` program for ideas.

MISSION 5.4 Change the Mission 5.3 program you wrote so that when it reaches zero distance, the IBot begins moving back and forth again, in increasing lengths. Save the program as `CH05M4.ASM` and run it on the `Field` Locale. Here are the basic tasks:

1 Turn the IBot to face east.
1a Put the initial distance to move (such as 20 units) into a register.
2 Make the IBot move a certain distance you choose.
3 Turn the IBot 180 degrees.

3a Decrement the distance to move.
4 Loop back to step 2 if the distance to move is greater than zero.
5 Otherwise, increment the distance to move.
6 Add the rest of the steps you need for the program. If you need help, see the `CH05DEC.ASM` and `CH05INC.ASM` programs for ideas.

6 - Coprocessors and Ports

In this chapter you'll learn how programs use IntelliBots coprocessors and how programs can read and use information from IntelliBots ports. Here are the main topics in this chapter:

- How Coprocessors Work
- How Ports Work
- Scanning
- Checking Object IDs
- IBot Offense
- Checking Offense Ports

How Coprocessors Work

Computer Coprocessors

A *coprocessor* is a specialized type of CPU, in addition to the main CPU, that performs specific tasks. Many computers today are equipped with coprocessors that help share the computer's data processing load. For example, math coprocessors do intensive calculations, while graphics coprocessors handle tasks for drawing and displaying graphics.

Each coprocessor does its assigned tasks while the main CPU is working on its own tasks. This is called *multiprocessing mode*, because multiple tasks are carried out at the same time. In IntelliBots, multiprocessor mode can be turned on or off. By default, IntelliBots uses single-processor mode: when a coprocessor instruction executes, the main processor waits until the coprocessor has finished its task. More information on multiprocessing will be available in *Course 2: Intermediate Concepts*.

IntelliBots Coprocessors

The IntelliBots program uses *simulated* coprocessors to turn and move the IBot. You have already used some of these instructions, such as:

```
COPR    #ChassisMove,#10     ;move IBot 10 squares
COPR    #ChassisTurn,#90     ;turn IBot chassis 90 degrees
```

```
COPR    #TurretTurn,#45      ;turn turret 45 degrees
```

`COPR` stands for COPRocessor. Below are other types of IntelliBots coprocessor commands that are described in this chapter. For a complete list of IBot coprocessor commands, see the *Programmer's Quick Reference*.

Name	Description
`ChassisScan`	Chassis scan (low-level)
`TurretScan`	Turret scan (high-level)
`OffenseCannon`	Fire cannon
`OffenseLaser`	Fire laser

How Ports Work

Computer Ports

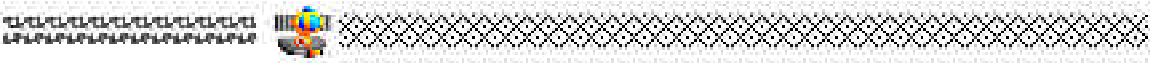
A *port* is a special memory location that communicates with hardware devices. Two types of ports often found in today's computers are serial ports and parallel ports. A serial port, such as a modem port, sends and receives data "serially", or one bit at a time in a data stream. A parallel port, such as a printer port, sends and receives data in a parallel fashion, or multiple bits at a time in a wider data stream.

IntelliBots Ports

IBots can find out information about their environment by reading data stored in IntelliBots *ports*. Here are the IntelliBots ports you'll learn about in this chapter:

Port Name	Description
`ScanDistance`	Distance (in Locale units) to the last object scanned.
`ScanObject`	ID (identification) of the last object scanned.
`Shells`	Number of cannon shells available to the IBot. Each IBot begins a Mission with at least 10 shells and as many as 10 shells per opponent.
`LaserTemp`	Current temperature (in heat units) of the IBot's laser. A value of -1 indicates the laser will no longer work because it has melted or has been disabled.
`CannonTemp`	Current temperature (in heat units) of the IBot's cannon. A value of _-1 indicates the cannon will no longer work because it has melted.

You can also write data to certain ports as explained in later chapters. For a complete list of IntelliBots ports and the read/write status for each see the *Programmer's Quick Reference*.



IBot Scanning

Each IBot can scan across the Locale, identifying objects such as IBots, goals, and terrain. There are two ways an IBot can scan: with its chassis or with its turret.

Chassis Scanning

A chassis scan looks for objects in a path that's as wide as the IBot. The chassis scan can detect low or high objects, but only at short distances. Scanning can help the IBot avoid running into damaging objects, as well as find other IBots or goals. The chassis scan detects the *first object* it encounters.

Chassis scanning uses the `#ChassisScan` coprocessor command:

```
COPR    #ChassisScan,<angle>
```

The `<angle>` is the number of degrees to the right of the current chassis direction that the scan should occur. If `<angle>` is zero, the chassis scans straight ahead, in the direction the chassis is currently facing. If `<angle>` is negative, the IBot scans to the left. (You don't have to turn the IBot chassis to point in the direction you want to scan.)

The diagrams below show how chassis scanning works. In Figure 6-1, the IBot scans straight ahead and finds the `PowerNatural` object; in Figure 6-B, it scans 45 degrees to the right and finds `HighCover`; and in Figure 6-C, it scans 45 degrees to the left and finds `LowCover`. Note: In actual Missions, scans are invisible.

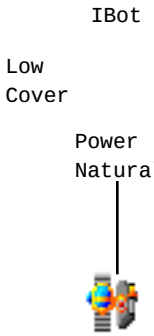


Figure 6-1: Chassis scan, ahead (COPR #ChassisScan,#0)

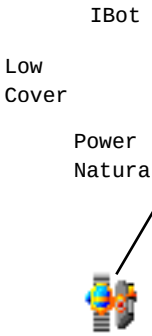


Figure 6-2: Chassis scan to the right (COPR #ChassisScan,#45)

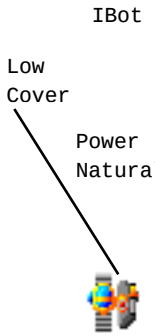


Figure 6-3: Chassis scan to the left (COPR #ChassisScan,#-45)

A chassis scan can also use an angle from a register:

```
MOVE    #90,R2              ;set scan angle to 90
COPR    #ChassisScan,R2     ;scan at angle contained in R2
```

ScanObject Port

When the scan finds an object, the object's *ID* is stored in the ScanObject port. To retrieve the ID of scanned object from the ScanObject port, use the GETP command, as shown below.

```
GETP    #ScanObject,R0      ;get scanned object's ID, put in R0
```

In this example, R0 is the register that receives the port data; you can use any register from R0 to R6. Once the port data is in a register, you can compare the data to a known object.

ScanDistance Port

When the scan finds an object, the distance to the object is stored in the ScanDistance port. To retrieve the *distance* to the scanned object from the ScanDistance port, you also use the GETP command, as shown below.

```
GETP    #ScanDistance,R1    ;get distance to scanned obj. into R1
```

In this example, R1 receives the port data; you can use any register from R0 to R6. However, *do not* use the same register you used for the ScanObject port. If you do, the ScanDistance data will replace the ScanObject data, because a register can hold only one value at a time.

The example below shows how the IBot scans 45 degrees to the right of the current chassis direction and gets the scanned object ID and distance.

```
COPR    #ChassisScan,#45   ;scan 45 degrees right of chassis dir.
GETP    #ScanObject,R0     ;get scanned object ID, put it in R0
GETP    #ScanDistance,R1   ;get scanned object distance, put in R1
```

Turret Scanning

Turret scanning scans in the direction the IBot turret is pointing. It detects only high objects, such as IBots, high goals, or high terrain. Turret scanning uses the `TurretScan` command. The second operand is *always zero.*

```
COPR    #TurretScan,#0      ;scan with turret
```

The turret scanning beam is one unit wide. Like the chassis scan, the turret scan stores the scanned object's ID in the `ScanObject` port, and the distance to the scanned object in the `ScanDistance` port. The scan finds the part of the object that is *closest* and returns that distance. The turret scan has a greater range than the chassis scan but loses accuracy at long distances.

Remember that the turret scan finds only high objects. In the diagram below, the turret scan looks past the `PowerNatural` terrain (because it's a low terrain) and finds the enemy IBot.

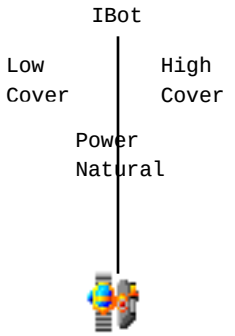


Figure 6-4: Turret scan
(`COPR #TurretScan,#0`)

To get information about the object scanned by the turret, use the same `GETP` commands as for the chassis scan. In the example below, the IBot does a turret scan and retrieves the scanned object ID and distance.

```
COPR    #TurretScan,#0     ;do turret scan
GETP    #ScanObject,R0     ;get scanned object ID, put it in R0
GETP    #ScanDistance,R1   ;get scanned object distance, put in R1
```

Turning the Turret You can turn the IBot turret by using the `TurretTurn` or `TurretTurnTo` coprocessor commands. In the example below, the `COPR #TurretTurn,#5` command turns the turret 5 degrees to the right of its current direction. (To turn the turret left, use a negative value.)

```
COPR    #TurretTurn,#5       ;turn turret 5 degrees to right
```

The `TurretTurnTo` command turns the turret to an *absolute* compass direction. In the example below, the `COPR #TurretTurnTo,#180` command turns the turret until it reaches 180 degrees (south), no matter where the turret was pointing before. (If the turret was already at 180 degrees, it doesn't turn.) This command will automatically determine and turn the turret in the most efficient direction.

```
COPR    #TurretTurnTo,#180   ;turn turret to 180-deg compass pt.
```

Checking Object IDs

Once the `GETP` command stores the scanned object ID in a register, your IBot needs to know how to identify that object ID. Object IDs are described in the chart below.

Damage to IBot shows what damage the IBot receives when it collides with the object. The types of damage are: light, medium, heavy, light power, medium power, and heavy power.

Hit By/Changes To shows what the object converts to when it is hit by an IBot, a laser, or a shell.

Object ID	Lo/Hi	Damage to IBot	Hit By >> Changes To ...
#Empty	low	none	IBot or laser >> no change; shell >> light obstruction
#Barrier	high	heavy power + heavy	No change. Locale borders are defined as barriers.
#LowCover	low	none	IBot >> Empty; laser >> no change; shell >> light obstruction
#HighCover	high	none	IBot >> Empty; laser >> low cover; shell >> light obstruction
#LightNatural	low	light	IBot >> no change; laser >> no change; shell >> light obstruction
#LightObstr	low	light	No change. A shell explodes into rubble, which is a light obstruction.
#MediumNatural	high	medium	IBot or laser >> light natural; shell >> light obstr.
#MediumObstr	high	medium	Any >> light obstruction
#HeavyNatural	high	heavy	IBot or laser >> light natural; shell >> light obstr.
#HeavyObstr	high	heavy	any >> light obstruction
#PowerNatural	low	light power	No change
#PowerObstr	low	med. power + heavy	Any >> light obstruction
#IBot1	high	med. power + heavy	No change. IBot1 through IBot4 match the order of IBots in Select IBots list (first IBot in list is IBot1, next IBot in list is IBot2, etc.).
#IBot2	"	"	"
#IBot3	"	"	"
#IBot4	"	"	"
#LowGoal	low	none	Any >> no change, but Mission terminates
#HighGoal	high	none	Any >> no change, but Mission terminates

In the example below, CMP compares the scanned object in R0 to a Barrier.

```
GETP    #ScanObject,R0      ;get scanned object ID, put it in R0
CMP     #Barrier,R0         ;is object a Barrier?
```

Scanning Review

You should be able to answer these questions about scanning:

- What type of scanning detects high objects?
- What type of scanning detects low objects?
- What type of scanning covers longer distances?
- What ports contain scan information, and how are they used?
- What kind of damage results when two IBots collide?

Taking Action

After you compare the scanned object to an object ID, you can jump to another part of your program to take action. In the example below, the IBot scans for a Barrier. When it finds one, it moves next to it (one unit away from it) but does not run into it. (Moving next to a Barrier can protect the IBot from enemy scans on that side.)

```
    MOVE    #0,R2               ;start chassis scan angle at zero
Scan
    COPR    #ChassisScan,R2     ;do chassis scan (zero degrees)
    GETP    #ScanObject,R0      ;get scanned object ID, put in R0
    GETP    #ScanDistance,R1    ;get dist. to scanned obj., put in R1
    CMP     #Barrier,R0         ;is it a barrier?
    JE      MoveNextTo          ;   -yes, go move next to it
    ADD     #5,R2               ;   -no, add 5 degrees to scan angle
    JUMP    Scan                ;    and go scan again
MoveNextTo
    COPR    #ChassisTurnTo,R2   ;turn chassis in direction of scan
    DEC     R1                  ;decrease R1 (1 less than scan dist)
    COPR    #ChassisMove,R1     ;move new R1 distance (by barrier)
```

Checking Multiple Object IDs

Objects are sorted by their ID, so you can check for a *group* of objects at once. To do this, you can use the JG or JL instructions.

The following example looks for any IBot. First it skips all objects less than IBot1, then it skips objects greater than.

```
Scan
    COPR    #ChassisScan,R2     ;do chassis scan (zero degrees)
    GETP    #ScanObject,R0      ;get scanned object ID, put in R0
    GETP    #ScanDistance,R1    ;get dist. to scanned obj., put in R1
    CMP     #IBot1,R0           ;is it < IBot1?
    JL      Scan                ;   -yes (it's less), so go scan
    CMP     #IBotLast,R0        ;is it > IBotLast?
    JG      Scan                ;   -yes (it's more), go scan again
Attach                          ;go attack it
    ...
```

In *Chapter 7: Computer Numbers and Bit Testing* you will learn how to use computer bits to check for multiple conditions more quickly.

IBot Offense

Each IBot has two kinds of offense; a laser and a cannon. They are always fired in the direction of the turret. An IBot's laser and cannon are only as accurate as the programming instructions you give. (These can be disabled for all IBots, in Preferences.)

Laser

With the laser, the IBot can hit only high objects. At longer distances, the laser causes less damage unless you increase its power. You can set the laser power from zero (no damage) to 50, or *LaserMaxPower*, (maximum damage). Each time the laser is fired it heats up, so it may melt if fired too often or at high power without letting it cool. Below is an example of the command to fire the laser.

```
COPR    #OffenseLaser,#LaserMaxPower ;fire at max. strength
```

Cannon

The cannon can hit high or low objects. Each shell radiates damage out from the point where it explodes. An IBot begins a Mission with 10 cannon shells per opponent, with a minimum of 10 shells. Each time the cannon is fired it heats up, so it may melt if fired too often without letting it cool. Below is an example of the command to fire the cannon.

```
COPR    #OffenseCannon,R1    ;launch shell dist. (R1) to target
```

A shell travels only the distance you specify, then explodes at that point (unless that point is off the Locale).

Exercise 1: Attacking IBots

The exercise below scans for enemy IBots. If an IBot is found, the program jumps to an attack routine.

1. Open a new file.

2. Carefully type in the source code below, including the comments.

```
Scan
    COPR    #TurretScan,#0       ;do turret scan
    GETP    #ScanDistance,R1     ;get dist. to scanned obj., put in R1
    GETP    #ScanObject,R0       ;get scanned object ID, put in R0
    CMP     #IBot1,R0            ;is it an IBot?
    JL      TurnTurret           ;   -no, so turn the turret 5 degrees
    CMP     #IBotLast,R0         ;is it an IBot?
    JG      TurnTurret           ;   -no, so turn the turret 5 degrees
Attack
    COPR    #OffenseCannon,R1    ;launch shell dist. (R1) to target
    COPR    #OffenseLaser,#LaserMaxPower    ;fire at maximum strength
```

```
    COPR    #TurretTurn,#720     ;spin turret twice, let cannon cool[1]
    JUMP    Scan                 ;go scan again
TurnTurret
    COPR    #TurretTurn,#5       ;turn turret 5 degrees
    JUMP    Scan                 ;go scan again
```

3 Read all the comments so you know what the program does. For example, does this program attack terrain objects? Why or why not? And if an enemy IBot moved, would your IBot be able to scan it again?

4 Save the file as `CH06PORT.ASM`.

5 Assemble the file without errors, then run a Mission on the `Test` Locale. Select `CH06PORT.BOT` and `MOVSLOW1.BOT`, in that order.

The `MOVSLOW1.BOT` IBot moves slowly across the Locale. It self-destructs after your IBot attacks it several times.

Checking Offense Ports

Laser and Cannon Temperatures

Each time an IBot fires its laser or cannon, the laser or cannon temperature increases, then gradually decreases (cools off). Firing the laser or cannon too often without letting it cool causes a meltdown. A melted laser can keep firing, but produces no laser beam. A melted cannon can keep firing, but the shells travel zero distance, so they explode on the IBot firing them.

The laser meltdown point is 75, or *LaserMeltTemp*, heat units. Each firing increases the laser temperature by 1 to *LaserMaxPower* heat units (the value in the `#OffenseLaser` command). For example, if you fire the laser at 50 heat units, the laser temperature also increases by 50 heat units. The laser temperature is stored in the `LaserTemp` port, so you can check it with the `GETP` instruction:

[1] Later in this chapter, you will learn how to check the laser and cannon heat without having to spin the turret.

```
    GETP    #LaserTemp,R0       ;get laser temp, put in R0
```

The cannon meltdown point is 125, or *CannonMeltTemp*, heat units. Each time the cannon is fired, its heat increases by 110, or *CannonInc*, units. The current cannon temperature is stored in the CannonTemp port, so you can check it with the GETP instruction:

```
    GETP    #CannonTemp,R1      ;get cannon temp, put in R1
```

Avoiding Meltdown

To avoid meltdown, your program can check the heat of the cannon or laser. If the cannon is too hot, the laser can be checked while the cannon cools. If the laser is also too hot, the IBot can go scan again while it waits. Now you can change the CH06PORT.ASM program to check for laser and cannon heat.

1. Add this line before the Scan label:

```
    COPR    #ChassisMove,#55    ;get closer to center of the Locale
```

2. Change the Attack code segment in the CH06PORT.ASM program to look like this:

```
Attack
    GETP    #CannonTemp,R2      ;get current cannon heat, put in R2
    CMP     #CannonMeltTemp-CannonInc-1,R2
                                ;is it too hot for another shot?
    JG      Laser               ;   -yes, try laser instead
    COPR    #OffenseCannon,R1   ;launch shell dist. (R1) to target
    JUMP    Scan                ;go back and scan
```

   Notice that the cannon temperature is checked *before* the cannon is fired so that meltdown is avoided.

3. After the Attack routine, add the Laser routine:

```
Laser
    GETP    #LaserTemp,R2       ;get current laser heat, put in R2
    MOVE    #LaserMeltTemp-1,R1 ;get the maximum safe temperature
    SUB     R2,R1               ;subtract the current temperature
    CMP     #LaserMaxPower,R1   ;is laser power less than maximum?
    JL      Laser               ;   -if less, wait
    COPR    #OffenseLaser,R1    ;fire laser with R1 units of power
    JUMP    Scan                ;go back and scan
```

Notice that `#LaserMeltTemp-1` (one less than the melting temperature) is used as the maximum safe temperature. You can subtract or add a value from or to any constant by attaching a minus or plus value after it. You can even add or subtract constants from each other such as `#CannonMeltTemp-CannonInc-1`.

4 Save the file as `CH06PRT1.ASM` and assemble the file without errors.

5 Select the `CH06PRT1.BOT` and `MOVSLOW1.BOT` IBots, in that order, and run a Mission on the `Test` Locale.

Shell Supply

Each IBot begins a Mission with 10 cannon shells per opponent, but at least 10 shells. Your program can use `GETP` to check the `Shells` port for the number of shells the IBot has left.

```
    GETP    #Shells,R0          ;get number of shells left, put in R0
```

If your IBot tries to launch a shell when none remain, the `#OffenseCannon` instruction still executes, but no shell is launched.

In the example below, the program jumps to a `Laser` routine if there are no shells left.

```
Attack
    GETP    #Shells,R4          ;get number of shells left, put in R4
    CMP     #0,R4               ;any shells left?
    JE      Laser               ;no, try laser instead
    GETP    #CannonTemp,R2      ;get current cannon heat, put in R2
    CMP     #CannonMeltTemp-CannonInc-1,R2
                                ;is cannon heat greater than is safe?
    JG      Laser               ;   -it's too hot, try laser instead
    COPR    #OffenseCannon,R1   ;launch shell, R1 distance to target
    JUMP    Scan                ;go back and scan
```

Summary

CONCEPTS

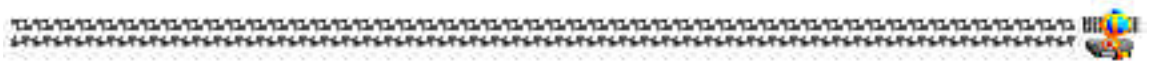
The following concepts were discussed in this chapter:

A) IntelliBots uses simulated coprocessors and ports to do command processing and setting and gathering of information.
B) Chassis scanning looks for objects at a given angle, using the `COPR #ChassisScan` command. The scan finds the first low or high object scanned.
C) Turret scanning looks for objects directly ahead of the turret, using the `COPR #TurretScan` command. The scan finds the first high object scanned.
D) The `GETP` instruction copies the data from the `ScanObject` or `ScanDistance` port into a register.
E) Each object can be identified by its unique scan identification.
F) `JG` and `JL` can be used to check a range of values after a `CMP` instruction.
G) The `LaserTemp` port contains the current laser temperature.
H) The `CannonTemp` port contains the current cannon temperature.
I) The `Shells` port contains the number of shells the IBot has remaining.

TERMS TO KNOW

After reading this chapter, you should be able to briefly define the terms below. If you need help, reread the chapter or see *Glossary* in this manual.

1) coprocessor; 2) port; 3) chassis scanning; 4) turret scanning;
5) scan distances; 6) object IDs; 7) `ScanDistance` port;
8) `ScanObject` port; 9) Shells port; 10) `LaserTemp` port;
11) `CannonTemp` port



Missions

MISSION 6.1 Change the `CH06PRT1.ASM` program so it only uses the laser to fire at the three `MOVSLOW` IBots. Save the program as `CH06M1.ASM`. Create and jump to a Laser routine similar to the Attack routine. Instead of deleting the instruction lines in Attack, turn them into comments by starting each of the lines with a semicolon. Run a Mission on the `Test` Locale; try to destroy the `MOVSLOW` IBots as soon as possible without melting the laser.

MISSION 6.2 Change the `CH06M1.ASM` program so it fires all its shells and then uses the laser. Save the program as `CH06M2.ASM`. Select the `MOVSLOW4`, `MOVSLOW5`, and `MOVSLOW6` IBots. Destroy these three IBots on the `Test` Locale as soon as possible, without melting the cannon or laser. (These `MOVSLOW` IBots do not self-destruct.)

MISSION 6.3 Change the `CH06M2.ASM` program to fire the shells and laser more efficiently. Save the program as `CH06M3.ASM`. Use the `Test` Locale and select your IBot, then the `MOVSLOW4` IBot. After waiting for the cannon or laser to cool, scan for the `MOVSLOW4` IBot again to see whether it has moved. If it has, find the IBot again and fire at it.

MISSION 6.4 Write a new program based on these tasks:

1) Scan around the Locale for a high goal.
1A) If a high goal is found, move to it.
2) If no high goal is found, move the IBot a few squares. You may want to check for Barriers, such as Locale borders, before moving. You may need to turn the IBot chassis before moving it.
3) Repeat tasks 1 through 2A in a loop.

You can borrow (copy and paste) elements from your `CH06PRT1.ASM` program and any of the `MOVSLOW` programs. Use the `Test` Locale; be *sure* to enable its goal. Run a Mission with only your IBot and find and move to the goal as quickly as possible. For more practice, you can try your IBot on any Locale with an enabled goal.