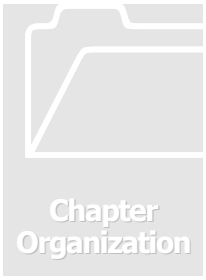
Chapter 6

Read this chapter if you know CM3-IDE well, and would like to use CM3-IDE for system development.

6. Development Recipes

Assuming that you are familiar with the CM3-IDE environment and the Modula-3 language, this chapter describes a number of recipes for building simple realistic systems applications.

A handful of complete programs illustrate the use of advanced facilities in CM3-IDE. You can find the sources for programs in this chapter in the Examples section of your CM3-IDE environment.



Using a simple automated bank teller scenario, **Robust Distributed Applications: Network Objects** on page 108 illustrates how to build distributed applications with Network Objects.

Client/Server Computing: Safe TCP/IP Interfaces on page 115 describes the safe, multi-platform, and multi-threaded TCP/IP interfaces. A Finger client and a simple HTTP server are described.

Taking Persistent Snapshots of Objects: Pickles on page 120 demonstrates how to transcribe objects onto an I/O stream.

Quick Comparison of Large Data: Fingerprints on page 122 outlines how to take fingerprints of large data structures, and use the fingerprints to compare the data structures efficiently.

Portable Operating System Interfaces on page 124 illustrates the use of portable interfaces for operating system services, such as: file systems access, process management, thread creation, and environment variables. A complete and portable command-shell program is used as a demonstration.

Dynamic Web Applications: the Web Server Toolkit on page 135 outlines a simple contact database program based on the web toolkit.

Interacting with C Programs on page 137 shows how to have your code call C programs or be called by C programs. Examples illustrate the integration of C source code and libraries into CM3-IDE.

6.1 Robust Distributed Applications: Network Objects

Network Objects allows an object to be handed to another process in such a way that the process receiving the object can operate on it as if it were local. The holder of a remote object can freely invoke operations on that object just as if it had created that object locally.

Further, it can pass the object to other processes. Thus, the Network Objects system allows the development of not just simple client/server applications, but more general multi-tiered distributed applications.

When a program calls another through Network objects, we refer to the caller as *the client*, and the callee as *the server*. In the context of network objects, the names client and server signify roles in a particular interaction—a server may in fact be a client of another server.

The contract between the client and the server is defined by a *common interface*.

Here we describe a simple automated bank teller program as an example by outlining each component: the interface, the client and the server.

6.1.1 The Common Interface

The `Bank` interface defines the common contract between client and server in our example.

`NetObj` is the primary interface for building network object applications. `NetObj.Error` and `Thread.Alerted` may be raised by network object operations. A `Bank.T` is a network object which supports the operation `findAccount`, which returns a `Bank.Account` object. Type `Bank.Account` supports operations `deposit`, `withdraw` and `get_balance`.

Network object operations can raise user-defined exceptions such as `BadAmount`, and `InsufficientFunds`.

```
INTERFACE Bank;
IMPORT NetObj;
FROM NetObj IMPORT Error;
FROM Thread IMPORT Alerted;
TYPE
   T = NetObj.T OBJECT METHODS
      findAccount (acct: AcctNum): Account
         RAISES {Alerted, Error};
   END;
TYPE
   Account = NetObj.T OBJECT METHODS
      deposit (amount: REAL)
         RAISES {BadAmount, Alerted, Error};
      withdraw (amount: REAL) RAISES {BadAmount,
         InsufficientFunds, Alerted, Error};
      get_balance (): REAL RAISES {Alerted, Error};
   END;

TYPE
   AcctNum = [1..100];

EXCEPTION
   BadAmount;
   InsufficientFunds;
END Bank.
```

A simple makefile instructs CM3-IDE that `Bank.T` and `Bank.Account` are network objects. CM3-IDE will generate the required stubs automatically as part of this library, so a client or a server in this scenario may use `netobj-interface`.

Import netobj to bring in the network object libraries.

```
import("netobj")
```

For each network object type `I.T` you must call `netobj(I,T)`

```
interface("Bank")
netobj("Bank", "T")
netobj("Bank", "Account")
library("netobj-interface")
```

6.1.2 A Network Object Server

`NetObjServer` is a sample implementation of a network object server that exports an implementation of the `Bank` interface.

```
MODULE NetObjServer EXPORTS Main;
IMPORT Bank, NetObj, Thread;
IMPORT IO, Fmt;
```

`BankImpl` defines a full representation for the `Bank.T` network object.

```
TYPE
   BankImpl = Bank.T OBJECT
      accounts : ARRAY Bank.AcctNum OF Account;
   OVERRIDES
      findAccount := FindAccount;
   END;
```

Find an account in the table of accounts:

```
PROCEDURE FindAccount (self: BankImpl;
                       acct: Bank.AcctNum
                      ): Bank.Account =
BEGIN
   RETURN self.accounts[acct];
END FindAccount;
```

For `Bank.Account` network objects, `Bank.Account` uses a `MUTEX` to synchronize access to its balance. It also implements the operations `deposit`, `withdraw`, and `get_balance`.

```
TYPE
   Account = Bank.Account OBJECT
      lock : MUTEX;
      balance : REAL := 0.0;
   OVERRIDES
      deposit := Deposit;
      withdraw := Withdraw; (* not included *)
      get_balance := Balance; (* not included *)
   END;
```

Deposit the money, making sure to serialize access with others trying to operate on this account.

```
PROCEDURE Deposit (self: Account; amount: REAL)
   RAISES {Bank.BadAmount} =
BEGIN
   IF amount < 0.0
   THEN RAISE Bank.BadAmount;
   END;
   LOCK self.lock DO
      self.balance := self.balance + amount;
   END;
END Deposit;
```

Withdraw the money, making sure to serialize access with others trying to operate on this account.

```
PROCEDURE Withdraw (self: Account; amount: REAL)
   RAISES {Bank.BadAmount,
           Bank.InsufficientFunds} =
BEGIN
   IF amount < 0.0
   THEN RAISE Bank.BadAmount;
   END;
   LOCK self.lock DO
      IF self.balance < amount
      THEN RAISE Bank.InsufficientFunds
      END;
      self.balance := self.balance - amount;
   END;
END Withdraw;
```

Get the balance, making sure to serialize access with others trying to operate on this account.

```
PROCEDURE Balance (self: Account): REAL =
BEGIN
   LOCK self.lock DO
      RETURN self.balance;
   END;
END Balance;
```

Create a new bank by instantiating all the account objects.

```
PROCEDURE NewBank () : BankImpl =
VAR b := NEW (BankImpl);
BEGIN
   FOR i := FIRST (b.accounts) TO LAST (b.accounts) DO
      b.accounts[i] :=
         NEW (Account, lock := NEW (MUTEX));
   END;
   RETURN b;
END NewBank;
```

Print a summary of all the active accounts, i.e., ones that have a positive balance.

```
PROCEDURE PrintSummary() =
BEGIN
   IO.Put (BankName & ": active account information\n");
   FOR i := FIRST(bank.accounts) TO LAST(bank.accounts) DO
      IF bank.accounts[i].balance > 0.0 THEN
         IO.Put (Fmt.Int(i) & ".......$" &
            Fmt.Real(bank.accounts[i].balance) & "\n");
      END;
   END;
END PrintSummary;
```

Finally, the server's global variables and main body. The main body prints the summaries for accounts every 60 seconds. Since the network objects runtime forks and manages threads to handle incoming calls, the server can simply loop, printing its summary.

```
CONST
   BankName = "LastNationalBank";
VAR
   bank := NewBank();
BEGIN
   IO.Put ("Starting bank server.\n");
   TRY
      (* Export the bank object under "LastNationalBank". *)
      NetObj.Export (BankName, bank);
      IO.Put ("Bank server was exported as " &
         BankName & "\n");
      LOOP
         Thread.Pause (60.0D0);
         PrintSummary();
      END;
   EXCEPT (* If there is a problem, print an error and exit. *)
    | NetObj.Error =>
         IO.Put ("A network object failure occured.\n");
    | Thread.Alerted => IO.Put ("Thread was alerted.\n");
   END;
END NetObjServer.
```

The makefile for the server is simple. Note that the server must import the library defining the common interface. In this case, it's called `netobj-interface`.

```
import("netobj")
import("netobj-interface") % the common interface
implementation("NetObjServer")
program("netobj-server")
```

6.1.3 A Network Object Client

`NetObjClient` is a sample implementation of a network object client.

```
MODULE NetObjClient EXPORTS Main;
IMPORT Bank, NetObj, Thread;
IMPORT IO, Fmt, Scan, Text, FloatMode, Lex;
VAR
   bank: Bank.T;
   acctnum: Bank.AcctNum;
   acct: Bank.Account := NIL;
   cmd: TEXT;
```

Print a prompt on the screen and asks for input from the user. If the current account is set, it will display the current account and the available balance.

```
PROCEDURE Prompt(txt: TEXT): TEXT RAISES {IO.Error} =
BEGIN
   TRY
      IF acct # NIL THEN
         IO.Put ("\n[acct:" & Fmt.Int(acctnum) &
            ", balance: $" &
         Fmt.Real (acct.get_balance()) & "] ");
      END;
   EXCEPT
      ELSE (* since it is only a prompt, we ignore all exceptions *)
   END;
   IO.Put (txt & " : ");
   RETURN IO.GetLine();
END Prompt;

EXCEPTION
   InvalidAccount;
   Quit;
```

This client will take input commands and make calls to network objects. As you can see, most of the work is in the reading of input from the user!

```
CONST
   BankName : TEXT = "LastNationalBank";
BEGIN
   IO.Put ("Welcome to " & BankName & "\n");
   IO.Put ("Connecting to bank server...");
   TRY
      bank := NetObj.Import (BankName);
   EXCEPT
      IO.Put ("done.\n");
      IO.Put ("Bank Teller Client Started...\n");
      IO.Put ("Valid commands are: \n" &
         " account : set a current account for further " &
                     "transactions\n" &
         " deposit : deposit into current account \n" &
         " withdraw: withdraw from the current account\n" &
         " balance : print balance for the current account\n" &
         " quit    : quit bank teller client\n");
      IO.Put ("\n");
```

```
        LOOP
           TRY
              cmd := Prompt("Command: ");
              IF Text.Equal (cmd, "account")
              THEN (* new account *)
                 WITH input = Scan.Int(Prompt("account number")) DO
                    IF input < FIRST(Bank.AcctNum) OR
                       input > LAST(Bank.AcctNum)
                    THEN
                       RAISE InvalidAccount;
                    END;
                    acct := bank.findAccount(input);
                    acctnum := input;
                 END;
              ELSIF Text.Equal(cmd, "deposit")
              THEN (* deposit *)
                 IF acct = NIL THEN RAISE InvalidAccount END;
                 WITH amount = Scan.Real(Prompt(" amount")) DO
                    acct.deposit(amount);
                 END;
              ELSIF Text.Equal(cmd, "withdraw")
              THEN (* withdraw *)
                 IF acct = NIL THEN RAISE InvalidAccount END;
                 WITH amount = Scan.Real(Prompt(" amount")) DO
                    acct.withdraw(amount);
                 END;
              ELSIF Text.Equal(cmd, "balance")
              THEN (* get balance *)
                 IF acct = NIL THEN RAISE InvalidAccount END;
                 IO.Put ("Balance is " &
                          Fmt.Real(acct.get_balance()) & "\n");
              ELSIF Text.Equal(cmd, "quit")
              THEN (* quit by raising the "Quit" exception. *)
                 RAISE Quit;
              ELSE (* invalid command *)
                 IO.Put ("Valid commands are: account, " &
                    "deposit, withdraw, balance, and quit.\n");
              END;
           EXCEPT
            | Bank.BadAmount => IO.Put("Can't withdraw or " &
                 "deposit negative amounts.\n");
            | InvalidAccount => IO.Put ("Select an account " &
                 "in the range [" &
                 Fmt.Int(FIRST(Bank.AcctNum)) & ".." &
                 Fmt.Int(LAST(Bank.AcctNum)) & "] first.\n");
            | FloatMode.Trap, Lex.Error => IO.Put ("Cannot " &
                 "convert the number as specified.\n");
            | Bank.InsufficientFunds => IO.Put ("Insufficient " &
                 "funds available to perform this transaction\n");
           END;
        END;
     EXCEPT
      | NetObj.Error =>
           IO.Put ("A network object error occured\n");
      | Thread.Alerted => IO.Put ("A thread was alerted\n");
      | IO.Error, Quit => IO.Put ("Goodbye.\n");
     END;
  END NetObjClient.
```

Finally, the makefile for a client:

```
import("netobj")
import("netobj-interface") % the common interface
implementation("NetObjClient")
program("netobj-client")
```

6.2 Client/Server Computing: Safe TCP/IP Interfaces

Using CM3-IDE's safe TCP/IP interfaces, it is easy to program multi-threaded TCP clients and servers. Two examples, a Finger client and a simple HTTP server illustrate the use of the TCP interfaces. These same programs will work with Unix sockets or the Windows Winsock libraries without requiring source changes.

6.2.1 A TCP/IP Client: Finger

`Finger` is a simple program which introduces TCP client services. It also shows you how to bind TCP/IP connections to input and output streams.

```
MODULE Finger EXPORTS Main;
IMPORT TCP, IP, ConnRW;
IMPORT IO, Params;
FROM Text IMPORT FindChar, Sub, Length;
IMPORT Thread; <* FATAL Thread.Alerted *>
```

Common Constants and Variables.

Port 79 is the internet standard for the finger socket port. Variables `user`, and `host` are used by code in this module.

```
CONST
   FingerPort = 79;
VAR
   user := "";
   host := "localhost";
   addr : IP.Address;
```

Command Line Parameters.

Exception `Problem` is used to flag problems with the parameters.

```
EXCEPTION
   Problem;
```

Parse the `user` and `host` from arguments. Raise `Problem` if they're bad.

```
PROCEDURE GetUserHost() RAISES {Problem} =
BEGIN
   IF Params.Count # 2 THEN
      IO.Put ("Syntax: finger user@host\n");
      RAISE Problem;
   END
   IF Params.Count = 2 THEN
      user := Params.Get(1)
   END;
   WITH at = FindChar(user, '@') DO
      IF at = -1 THEN
         host := "localhost";
      ELSE
         host := Sub (user, at+1, LAST(INTEGER));
         user := Sub (user, 0, at);
      END;
   END;
END GetUserHost;
```

Main Implementation.

```
BEGIN
   TRY
      (* Get the values for user and host: *)
      GetUserHost();
      IO.Put ("(Checking for " & user &
         " finger information on host" & host & ")\n");
      (* Lookup host by name: *)
      IF NOT IP.GetHostByName (host, addr) THEN
         IO.Put ("Could not find hostname " &
                  host & "\n");
         RAISE Problem;
      END;
      (* Connect to the endpoint at port 79 of host.
         Get a reader and a writer to that port. *)
      VAR
         endpoint := IP.Endpoint {addr, FingerPort};
         service := TCP.Connect(endpoint);
         rd := ConnRW.NewRd(service);
         wr := ConnRW.NewWr(service);
      BEGIN
         (* Send the user name to the writer; read the
            whole response until EOF from the reader *)
         IO.Put (user & "\n", wr);
         WHILE NOT IO.EOF (rd) DO
            IO.Put (IO.GetLine(rd) & "\n")
         END
      END
      (* Check for possible errors. *)
   EXCEPT
    | IO.Error, IP.Error =>
         IO.Put ("Problem communicating with " &
                  host & "...\n");
    | Problem => (* Error has already been printed, do
                    nothing. *)
   END
END Finger.
```

6.2.2 A TCP/IP Server: HTTPD

The program HTTPD implements a simple HTTP server by using the portable TCP/IP interfaces. The basic outline of the program is simple: After getting a connector, loop and do the following:

1. Use `TCP.Accept` to get a new service.

2. Get a reader and a writer to the service via the `ConnRW` interface.

3. Use `Lex.Match` to ensure that the requests start with a "`GET`".

4. The rest of the input from the reader until the end of the line is the path requested by the web browser.

5. Given a path requested by a "`GET`" message, look in the current directory of your file system for the file in question. So, the URL
   ```
   http://localhost:80/welcome.html
   ```
   maps to the following HTTP request to the server running on port 80 of the machine "`localhost`":
   ```
   GET /welcome.html
   ```
   which maps to the file `welcome.html` in your file system.

6. Open the file, and read its contents.

7. Write the contents to the writer that is hooked up to the network connection. Flush the writer upon completion.

8. Make sure to close the reader, the writer, and the server connection at the bottom of the loop.

Here is the implementation for `HTTPD`. Review the `TCP` and `IP` interfaces for more information regarding the TCP/IP calls.

```
MODULE HTTPD EXPORTS Main;
IMPORT TCP, IP, ConnRW;
IMPORT Rd, Wr, IO, Lex, FileRd, RdCopy;
IMPORT Thread, OSError, Text, Params, Process, Pathname;
```

Use `http://hostname:80/` to access this server:

```
CONST
   HTTP_Port = 80;
PROCEDURE Error (wr: Wr.T; msg: TEXT)
   RAISES {Thread.Alerted, Wr.Failure} =
BEGIN
   Wr.PutText (wr, "400 " & msg );
   Wr.Flush (wr);
END Error;
```

Create an endpoint on the `HTTP_Port`. Get a connector for the end point, and loop:

- Use `TCP.Accept` to wait for a new connection that can handle calls.
- Create a reader and a writer to the connection.
- Look for a `GET`, and then a path for the request. Parse pathname and print it. If there is a request for root, return a welcome string, otherwise find the file residing in a subdirectory.

Of course, catch all the possible exceptions.

```
VAR
   endpoint := IP.Endpoint {IP.GetHostAddr(), HTTP_Port};
   connector: TCP.Connector;
   server: TCP.T;
   rd: Rd.T; wr: Wr.T;
   path: TEXT;
BEGIN
   TRY
      connector := TCP.NewConnector(endpoint);
      LOOP
         server := TCP.Accept(connector);
         rd := ConnRW.NewRd(server);
         wr := ConnRW.NewWr(server);
         TRY
            TRY
               Lex.Match (rd, "GET ");
               path := Lex.Scan (rd);
               IO.Put ("path=" & path & "\n");
               IO.Put (Rd.GetLine(rd) & "\n");
               IF Text.Equal (path, "/") THEN
                  Wr.PutText (wr,
                     "<H1>Welcome to our web server!" &
                     "</H1>Try <a href=welcome.html>" &
                     "this link" & "</a>.\n");
               ELSE
                  WITH rd = FileRd.Open (Text.Sub (path,
                                1, Text.Length(path))) DO
                     TRY
                        RdCopy.ToWriter(rd,wr);
                     FINALLY
                        Rd.Close(rd);
                     END;
                  END
               END;
               Wr.Flush (wr); (* so the browser can see
                                 the results. *)
            EXCEPT
             | Lex.Error => Error (wr,
                  "Only GET methods are supported\n");
             | OSError.E => Error (wr,
                  "File not found or no permission.\n");
             | Rd.EndOfFile => Error (wr,
                  "Request terminated prematurely.\n");
            END;
         FINALLY (* clean up on your way out. *)
            Rd.Close (rd);
            Wr.Close (wr);
            TCP.Close (server);
         END;
      END
   EXCEPT
    | Thread.Alerted => IO.Put ("Thread was alerted\n");
    | IP.Error => IO.Put ("IP error\n");
    | Rd.Failure, Wr.Failure =>
         IO.Put ("Rd/Wr failure\n");
   END;
END HTTPD.
```

6.3 Taking Persistent Snapshots of Objects: Pickles

Pickles can be used to load and save the state of objects via I/O streams bound to disk files, network connections, or in-memory data. To learn more about pickles, browse the `Pickle` interface.

This program uses pickles to snapshot a copy of its internal database to disk, and load it later. The internal database is kept as a list of *atoms*. An atom is a unique representation for a text string.

```
MODULE PickleExample EXPORTS Main;
IMPORT Pickle, Wr, FileWr, Rd, FileRd;
IMPORT Atom, AtomList;
IMPORT Action, AtomActionTbl;
IMPORT Process, IO; <* FATAL IO.Error *>
```

Import the `Pickle` interface to take snapshots of objects and turn the snapshots back into live objects, `Wr` and `FileWr` to write snapshots to files, and `Rd`, and `FileRd` to read snapshots from files.

Import `Atom` and `AtomList` interfaces. An `Atom` is unique representation of a string, you can convert text to `Atom` and then compare it with other `Atom`s without using text operations.

Import the `Action` interface, defined in this package, and `AtomActionTbl`, a table mapping atoms to actions.

Atom List Operations. `Contains`, `Insert` and `Print` are utility functions which call `AtomList` operations.

Insert an element into the list.

```
PROCEDURE Insert (VAR list: AtomList.T; atom: Atom.T) =
BEGIN
   IF NOT AtomList.Member(list, atom) THEN
      list := AtomList.Cons (atom, list);
   END
END Insert;
```

Print out all elements of the list by iterating over its members.

```
PROCEDURE Print(x: AtomList.T) =
BEGIN
   WHILE x # NIL DO
      IO.Put (Atom.ToText (x.head) & " ");
      x := x.tail;
   END;
END Print;
```

Command Operations. Definition of what commands should do. `Actions` define initial values for the action table.

```
TYPE
   Commands = {Show, Quit, Reset, Help, Load, Save};

   Actions = ARRAY OF Action.T {
                Action.T { "show", Show},
                Action.T { "quit", Quit},
                Action.T { "reset", Reset},
                Action.T { "help", Help}
                Action.T { "load", Load}
                Action.T { "save", Save}};
```

Each procedure defines what each action should do. Note that `Actions` includes elements that happen to be procedures. Also that the `proc` field of `Action.T` is defined to be a `PROCEDURE()`, so we can assign any of `Quit`, `Rest`, `Help`, or `Show` to fields of `Actions`.

```
PROCEDURE  Quit() = BEGIN Process.Exit(0);  END Quit;
PROCEDURE Reset() = BEGIN input_set := NIL; END Reset;

PROCEDURE  Show() =
BEGIN
   Print(input_set);
   IO.Put ("\n");
END Show;

PROCEDURE Help() =
BEGIN
   IO.Put("Commands: show, reset, " &
          "help, quit, load, save.\n" &
          "Otherwise: insert into the list.\n");
END Help;
```

Procedures `Save` and `Load` use pickles to save and load the database.

```
CONST DB = "db";

PROCEDURE Save() =
   VAR wr := IO.OpenWrite(DB);
BEGIN
   Pickle.Write (wr, input_set);
   Wr.Close (wr);
END Save;

PROCEDURE Load() =
   VAR rd := IO.OpenRead (DB);
BEGIN
   input_set := Pickle.Read (rd);
   Rd.Close (rd);
END Load;
```

Main Program. The principal data in this program: `command_table` is an atom→action table; `input_set` is an atom list, containing all the elements that will be entered.

```
VAR
   command_table := NEW(AtomActionTbl.Default).init();
   input_set : AtomList.T := NIL;
BEGIN
   (* Initialize Commands. *)
   FOR x := FIRST(Actions) TO LAST(Actions) DO
      EVAL command_table.put(Atom.FromText (x.name), x);
   END;

   IO.Put ("Welcome to the atomic database.\n");
   IO.Put ("Try any of commands: show quit reset help.\n");
   IO.Put ("Any other string will be entered into " &
           "the database.\n\n");
```

Loop, get the user response from the command line. If it's a command, do it. Otherwise insert the command line into the `input_set`. If `atom` is in the `command_table` then run the corresponding `action`. Otherwise, `Insert` the `atom` into the `input_set`.

```
   LOOP
      IO.Put ("persistent atom-db > ");
      IF IO.EOF () THEN EXIT END;
      VAR
         cmd := IO.GetLine();
         atom := Atom.FromText(cmd);
         action: Action.T;
      BEGIN
         IF command_table.get(atom, action)
         THEN action.proc();
         ELSE Insert(input_set, atom);
         END;
      END;
   END;

END PickleExample.
```

6.4 Quick Comparison of Large Data: Fingerprints

You can use the `Fingerprint` interface to compare large amounts of data. Fingerprints can also be used for efficient comparison of complex object graphs.

The program `M3Compare` takes two file names from the command line and reports whether the files are the same or different. The program does not crash due to exceptions.

```
MODULE M3Compare EXPORTS Main;
IMPORT IO, Process, Fingerprint, Rd, Thread, Params;
```

Use `Fingerprint.FromText` to get a fingerprint of each file, then compare the finger prints.

```
PROCEDURE Compare (a, b: TEXT) =
   VAR aa, bb: TEXT;
BEGIN
   aa := Inhale (a);
   bb := Inhale (b);
   IF (aa = NIL) OR (bb = NIL) THEN
      (* already reported an error *)
   ELSIF Fingerprint.FromText(aa) =
         Fingerprint.FromText(bb)
   THEN
      IO.Put ("The files are the same.\n");
   ELSE
      IO.Put ("The files are different.\n");
   END;
END Compare;
```

Read a file and return its contents as text.

```
PROCEDURE Inhale (file: TEXT): TEXT =
   VAR rd: Rd.T; body: TEXT;
BEGIN
   rd := IO.OpenRead (file);
   IF (rd = NIL) THEN
      IO.Put ("\"" & file & "\" is not a file.\n");
      RETURN NIL;
   END;
   TRY
      body := Rd.GetText (rd, LAST (CARDINAL));
      Rd.Close (rd);
   EXCEPT Rd.Failure, Thread.Alerted =>
      IO.Put ("Unable to read \"" & file & "\".\n");
      RETURN NIL;
   END;
   RETURN body;
END Inhale;

BEGIN
   IF Params.Count # 3 THEN
      IO.Put ("syntax: m3compare <file1> <file2>\n");
      Process.Exit(2);
   END;
   Compare (Params.Get(1), Params.Get(2));
END M3Compare.
```

6.5 Portable Operating System Interfaces

Using the portable operating systems interfaces, you can write programs to get information about operating system facilities such as the files, directories, processes, paths, environment variables and command-line parameters. Following the interface specifications, you can write programs that do not depend on idiosyncrasies of different versions of Unix and Windows.

The program M3Sh, a simple command-line shell, operates like a normal DOS or Unix command shell, providing you with simple commands. Of course, m3sh does not depend on pre-processor macros.

```
MODULE M3sh EXPORTS Main;
```

M3sh is a simple shell utility which uses the safe, portable operating system interfaces. By using the portable interfaces, this program works on both Unix and Win32 platforms.

```
IMPORT Pathname, FS, IO, OSError;
IMPORT Stdio, RegularFile, Pipe;
IMPORT Process, Thread, Env, Params;
IMPORT FileWr, FileRd, Rd, Lex, Wr, Text, Atom, AtomList;
IMPORT TextRd, TextSeq;
```

Shell Commands. Command designates a name and an action procedure.

```
TYPE
   Command = RECORD
      name: TEXT;
      action: PROCEDURE (cmd: TEXT;
                         READONLY args: ARRAY OF TEXT
                        ): TEXT RAISES {OSError.E};
   END (* RECORD *);
```

Commands is an array of pre-defined Command designations for built-in shell commands. To allow aliasing of actions, multiple names may correspond to the same action.

```
CONST
   Commands = ARRAY OF Command {
      Command {"exit", exit},
      Command {"quit", exit},
      Command {"bye", exit},
      Command {"cd", chdir},
      Command {"chdir", chdir},
      Command {"dir", dir},
      Command {"ls", dir},
      Command {"pwd", pwd},
      Command {"directory", dir},
      Command {"type", type},
      Command {"cat", type},
      Command {"exec", exec},
```

```
        Command {"bg", background},
        Command {"help", help}};
```

Execute the shell command `cmd` with arguments `args`:

```
PROCEDURE ShellCommand(cmd: TEXT;
                       READONLY args: ARRAY OF TEXT
                      ): TEXT RAISES {OSError.E} =
```

Check to see if `cmd` is a built-in. If `cmd` is not a built-in, then try to execute it.

```
BEGIN
   FOR i := FIRST(Commands) TO LAST(Commands) DO
      IF Text.Equal (cmd, Commands[i].name) THEN
         RETURN Commands[i].action (cmd, args);
      END;
   END;
   RETURN Execute (cmd, args);
END ShellCommand;
```

Run an external command, returning the result as a text string. See the `Process` interface for more information.

```
PROCEDURE Execute(cmd: TEXT;
                  READONLY args: ARRAY OF TEXT
                 ): TEXT RAISES {OSError.E} =
   VAR hrChild, hwChild, hrSelf, hwSelf: Pipe.T;
   VAR result: TEXT := "";
BEGIN
   WITH full = FindExecutable(cmd) DO
      IF full # NIL THEN cmd := full; END;
   END;

   Pipe.Open(hr := hrChild, hw := hwSelf);
   Pipe.Open(hr := hrSelf, hw := hwChild);

   TRY
      WITH p = Process.Create (cmd, args, stdin := hrChild,
                stdout := hwChild, stderr := NIL) DO
         TRY
            TRY hrChild.close(); hwChild.close()
            EXCEPT OSError.E => (* skip *)
            END;
            (* Here is the actual writing and reading,
               conveniently performed using I/O streams. *)
            WITH wr = NEW(FileWr.T).init(hwSelf),
                 rd = NEW(FileRd.T).init(hrSelf) DO

               TRY Wr.Close(wr)
               EXCEPT Wr.Failure, Thread.Alerted => (*SKIP*)
               END;

               result := Rd.GetText(rd, LAST(INTEGER));

               TRY Rd.Close(rd)
               EXCEPT Rd.Failure, Thread.Alerted => (*SKIP*)
               END
            END;
         FINALLY EVAL Process.Wait(p);
         END
      END
   EXCEPT
    | Rd.Failure, Thread.Alerted => Error ("exec failed");
   END;
   RETURN result;
END Execute;
```

Check the number of arguments and raise `OSError.E` if the wrong number of arguments are being passed.

```
PROCEDURE ArgCount(READONLY args: ARRAY OF TEXT;
                   lo: CARDINAL;
                   hi: CARDINAL := LAST(INTEGER)
                  ) RAISES {OSError.E} =
BEGIN
   IF NUMBER(args) < lo THEN Error ("Too few args");
   ELSIF NUMBER(args) > hi THEN Error ("Too many args");
   END;
END ArgCount;
```

Given a string, procedure `Error` raises `OSError.E` with that string as a parameter.

```
PROCEDURE Error (name: TEXT) RAISES {OSError.E} =
VAR
   err := AtomList.List2(Atom.FromText(name),
                         Atom.FromText("m3sh error"));
BEGIN
   RAISE OSError.E(err);
END Error;
```

Built-in Commands. This section includes all the built-in shell commands, such as `dir` or `cd`.

```
PROCEDURE pwd(<*UNUSED*>cmd: TEXT;
              READONLY args: ARRAY OF TEXT
             ): TEXT RAISES {OSError.E} =
BEGIN
   ArgCount(args, 0, 0);
   RETURN Process.GetWorkingDirectory();
END pwd;

PROCEDURE dir(<*UNUSED*>cmd: TEXT;
              READONLY args: ARRAY OF TEXT
             ): TEXT RAISES {OSError.E} =
VAR
   dir: Pathname.T := ".";
   result: TEXT := "";
   name: TEXT;
   iter: FS.Iterator;
BEGIN
   ArgCount(args, lo := 0, hi := 1);
   IF NUMBER(args) > 0 THEN dir := args[0] END;
   IF NOT IsDirectory (dir) THEN
      Error (dir & " is not a directory");
   END;

   IO.Put ("Directory listing for " &
           FS.GetAbsolutePathname(dir) & "\n");
   iter := FS.Iterate (dir);
   WHILE iter.next (name) DO
      result := result & " " & name & "\n";
   END;
   iter.close();
   RETURN result;
END dir;

PROCEDURE chdir(<*UNUSED*>cmd: TEXT;
                READONLY args: ARRAY OF TEXT
               ): TEXT RAISES {OSError.E} =
BEGIN
   ArgCount(args, 1, 1);
   IF NOT IsDirectory (args[0]) THEN
      Error (args[0] & " is not a directory\n");
   END;
   Process.SetWorkingDirectory(args[0]);
   RETURN NIL;
END chdir;
```

Display the contents of a file or directory. If `arg[0]` is a file, return its contents. If `arg[0]` is a directory, prints its directory listing.

```
PROCEDURE type(cmd: TEXT;
               READONLY args: ARRAY OF TEXT
              ): TEXT RAISES {OSError.E} =
VAR rd: Rd.T;
BEGIN
   ArgCount(args, 1, 1);
   IF IsDirectory (args[0]) THEN
      Error (args[0] & " is a directory\n");
   ELSE
      TRY
         rd := FileRd.Open(args[0]);
         TRY RETURN Rd.GetText(rd, LAST(INTEGER));
         FINALLY Rd.Close(rd);
         END;
      EXCEPT
       | Rd.Failure, Thread.Alerted =>
           Error ("type could not read a file\n");
      END;
   END;
   <* ASSERT FALSE *>
END type;


PROCEDURE exit(<*UNUSED*>cmd: TEXT;
               READONLY args: ARRAY OF TEXT
              ): TEXT RAISES {OSError.E} =
BEGIN
   ArgCount(args, 0, 0);
   IO.Put ("Goodbye!\n");
   Process.Exit(0);
   <* ASSERT FALSE *>
END exit;


PROCEDURE exec(<*UNUSED*>cmd: TEXT;
               READONLY args: ARRAY OF TEXT
              ): TEXT RAISES {OSError.E} =
BEGIN
   ArgCount(args, 1);
   IO.Put ("The command is " & args[0] & "\n");
   RETURN Execute (args[0],
                   SUBARRAY(args,1, NUMBER(args)-1));
END exec;


PROCEDURE help(<*UNUSED*>cmd: TEXT;
               READONLY args: ARRAY OF TEXT
              ): TEXT RAISES {OSError.E} =
BEGIN
   ArgCount (args, 0, 0);
   RETURN HelpfulInfo();
END help;
```

Using Threads for the Background Command.

```
PROCEDURE background(<*UNUSED*>cmd: TEXT;
                     READONLY args: ARRAY OF TEXT
                    ): TEXT RAISES {OSError.E} =
VAR background_closure: BgClosure;
BEGIN
   ArgCount(args, 1);
   background_closure := NEW(BgClosure, cmd := args[0],
      args := NEW(REF ARRAY OF TEXT, NUMBER(args)-1));
   background_closure.args^ :=
      SUBARRAY(args, 1, NUMBER(args)-1);
   EVAL Thread.Fork (background_closure);
   RETURN NIL;
END background;

(* Closure for the background thread. *)
TYPE
   BgClosure = Thread.Closure OBJECT
      cmd: TEXT;
      args: REF ARRAY OF TEXT;
   OVERRIDES
      apply := BackgroundApply;
   END;

(* Work of the background thread. *)
PROCEDURE BackgroundApply (cl: BgClosure): REFANY =
BEGIN
   TRY
      RETURN Execute (cl.cmd, cl.args^);
   EXCEPT OSError.E => (* ignore background errors *)
   END;
   RETURN NIL;
END BackgroundApply;
```

PATH Navigation. Win32 and Unix use the PATH variable to define a list of directories to search for executables. Here we search for executables using the PATH as our guide.

Finds an executable program found by searching the directories contained in the PATH environment variable. PATH variable is looked up using the Env interface. To look up the separator for PATH, we need to find out what sort of system we are running. To do so, we check to see if Pathname uses / or \. (See also SearchPath.)

```
PROCEDURE FindExecutable (file: TEXT): TEXT =
VAR path := Env.Get ("PATH");
CONST UnixExts = ARRAY OF TEXT { NIL };
CONST WinExts = ARRAY OF TEXT {NIL,"exe","com","cmd","bat"};
VAR on_unix: BOOLEAN :=
                Text.Equal(Pathname.Join("","",NIL),"/");
BEGIN
   IF on_unix
   THEN RETURN SearchPath (file, path, ':', UnixExts);
   ELSE RETURN SearchPath (file, path, ';', WinExts);
   END;
END FindExecutable;
```

Return **TRUE** if the name corresponds to a file.

```
PROCEDURE IsFile (file: TEXT): BOOLEAN =
BEGIN
   TRY
      WITH stat = FS.Status (file) DO
         RETURN stat.type = RegularFile.FileType;
      END
   EXCEPT
    | OSError.E => RETURN FALSE;
   END
END IsFile;
```

Return TRUE if the name corresponds to a directory.

```
PROCEDURE IsDirectory (file: TEXT): BOOLEAN =
BEGIN
   TRY
      WITH stat = FS.Status (file) DO
         RETURN stat.type = FS.DirectoryFileType;
      END
   EXCEPT
    | OSError.E => RETURN FALSE;
   END
END IsDirectory;
```

Search the items passed in as part of path for the file.

```
PROCEDURE SearchPath (file, path: TEXT;
                      sep: CHAR;
                      READONLY exts: ARRAY OF TEXT
                     ): TEXT =
VAR
   dir, fn: TEXT;
   s0, s1, len: INTEGER;
   no_ext: BOOLEAN;
BEGIN
   IF IsFile (file) THEN RETURN file; END;
   no_ext := Text.Equal (file, Pathname.Base (file));

   (* First try the file without looking at the path. *)
   IF no_ext THEN
      FOR i := FIRST (exts) TO LAST (exts) DO
         fn := Pathname.Join (NIL, file, exts[i]);
         IF IsFile (fn) THEN RETURN fn; END;
      END;
   END;

   IF path = NIL THEN RETURN NIL; END;
   IF Pathname.Absolute (file) THEN RETURN NIL; END;

   (* Try the search path *)
   len := Text.Length (path); s0 := 0;
   WHILE (s0 < len) DO
      s1 := Text.FindChar (path, sep, s0);
      IF (s1 < 0) THEN s1 := len; END;
      IF (s0 < s1) THEN
         dir := Text.Sub (path, s0, s1 - s0);
         IF no_ext THEN
            FOR i := FIRST (exts) TO LAST (exts) DO
               fn := Pathname.Join (dir, file, exts[i]);
               IF IsFile (fn) THEN RETURN fn; END;
            END;
         ELSE
            fn := Pathname.Join (dir, file, NIL);
            IF IsFile (fn) THEN RETURN fn; END;
         END;
      END;
      s0 := s1 + 1;
   END;

   (* SearchPath failed. *)
   RETURN NIL;
END SearchPath;
```

The main program and utility procedures.

```
PROCEDURE HelpfulInfo(): TEXT =
CONST
   Msg = "m3sh: a simple portable shell for POSIX and" &
         "Win32 written in Modula-3\n" &
         "syntax: m3sh [-prompt string | -help]\n" &
         "commands:";
VAR
   result := Msg;
BEGIN
   FOR i := FIRST(Commands) TO LAST(Commands) DO
      result := result & " " & Commands[i].name;
   END;
   RETURN result & "\n";
END HelpfulInfo;

VAR
   prompt: TEXT := "m3sh";
```

Echo the prompt. Get a command. If the command is not null, then execute it, and print its results on the screen.

```
PROCEDURE ProcessCommand()
   RAISES {OSError.E, Rd.EndOfFile} =
VAR
   cmdname: TEXT;               (* name of the command *)
   cmdargs: REF ARRAY OF TEXT; (* arguments of the command *)
   result: TEXT;
BEGIN
   IO.Put(prompt & "> ");
   GetCommand(cmdname, cmdargs);
   IF cmdname = NIL THEN RETURN END;
   result := ShellCommand (cmdname, cmdargs^);
   IF result # NIL THEN IO.Put (result & "\n") END;
END ProcessCommand;
```

Read a command line; affect variables `name` and `args`. Set `name` and `args` to `NIL` if there is no input in this line. Raise `Rd.EndOfFile` if the end of file is reached.

```
PROCEDURE GetCommand (VAR name: TEXT;
                      VAR args: REF ARRAY OF TEXT
                     ) RAISES {Rd.EndOfFile} =
VAR
   cmd := NEW(TextSeq.T).init();
   rd: Rd.T;
BEGIN
   name := NIL; args := NIL;
   TRY
      (* Read a line and map it to the reader "rd". *)
      rd := TextRd.New(Rd.GetLine(Stdio.stdin));

      (* Tokenize the line into a sequence of strings. *)
      TRY WHILE NOT Rd.EOF(rd) DO
         Lex.Skip(rd);
         cmd.addhi(Lex.Scan(rd)); END;
      EXCEPT Rd.Failure => (* do nothing *)
      END;

      (* Turn the sequence into a (command, arguments) pair. *)
      IF cmd.size() = 0 THEN RETURN END;
      name := cmd.get(0);
      args := NEW(REF ARRAY OF TEXT, cmd.size()-1);
      FOR i := FIRST(args^) TO LAST(args^) DO
         args[i] := cmd.get(i+1);
      END;
   EXCEPT
    | Rd.Failure, Thread.Alerted =>
         IO.Put ("Problems in reading from input\n");
   END;
END GetCommand;
```

Print arguments to an `OSError.E`. Used by the main shell loop to print out errors.

```
PROCEDURE PrintError (al: AtomList.T) =
BEGIN
   WHILE al # NIL DO
      IO.Put (Atom.ToText(al.head) & ". ");
      al := al.tail;
   END;
   IO.Put ("\n");
END PrintError;
```

Check the command-line parameters.

```
PROCEDURE ProcessParams() =
BEGIN
   CASE Params.Count OF
    | 1 => RETURN;
    | 2 => IF Text.Equal(Params.Get(1), "-help") THEN
              IO.Put (HelpfulInfo());
              RETURN
           END;
    | 3 => IF Text.Equal(Params.Get(1), "-prompt") THEN
              prompt := Params.Get(2);
              RETURN
           END;
   ELSE (* skip *)
   END;
   IO.Put ("Incorrect or bad number of parameters." &
           " Try -help to get more info.\n");
   Process.Exit(10);
END ProcessParams;
```

The main loop.

```
BEGIN
   ProcessParams();
   LOOP
      TRY
         ProcessCommand();
      EXCEPT
       | OSError.E (e) => PrintError(e);
       | Rd.EndOfFile => EXIT;
      END;
   END;
END M3sh.
```

Further Information. To learn more about operating system interfaces see **CM3-IDE Interface Index** on page 143, or interface definition for:

- `Process` interface for process management
- `Thread` interface for creating threads or "lightweight processes"
- `FS` interface for access to files and directories
- `Pathname` interface for manipulating pathnames in a portable fashion
- `Env` interface for environment variables
- `Params` interface for command-line parameters
- `OSError` interface for handling operating system errors

6.6 Dynamic Web Applications: the Web Server Toolkit

The web server toolkit defines a framework for building dynamic web servers. The program `WebContact` illustrates a simple web-based application of a dynamic contact database.

```
MODULE WebContact EXPORTS Main;
IMPORT HTTPApp, HTTPControl, HTTPControlValue, App;
IMPORT Text, TextTextTbl;
FROM IO IMPORT Put;
```

Create two fields for names and e-mail addresses. Each is displayed on an automatically generated form, and the call-back procedures are run when the form is submitted.

```
VAR
   name, email: TEXT := "";
```

Define a text control, `name_value`, and its `Get` and `Set`, and `Default` operations.

```
VAR
   name_value := NEW(HTTPControlValue.TextValue,
                     leader := "<strong>Name: </strong>",
                     id := "name", set := SetName,
                     get := GetName,
                     setDefault := Default);

PROCEDURE SetName(<*UNUSED*>self: HTTPControlValue.TextValue;
                  val: TEXT;
                  <*UNUSED*>log: App.Log) =
BEGIN
   name := val;
   IF NOT db.get(name, email) THEN
      email := "";
   END;
END SetName;

PROCEDURE GetName(<*UNUSED*>self: HTTPControlValue.TextValue
                 ): TEXT =
BEGIN
   RETURN name;
END GetName;
```

Define another text control `email_value`, and its `Get`, `Set`, and `Default` operations.

```
VAR
   email_value := NEW(HTTPControlValue.TextValue,
                      leader := "<strong> Email:</strong>",
                      id := "email", set := SetEmail,
                      get := GetEmail,
                      setDefault := Default);

PROCEDURE GetEmail (self: HTTPControlValue.TextValue
                   ): TEXT =
BEGIN
   RETURN email;
END GetEmail;


PROCEDURE SetEmail (self: HTTPControlValue.TextValue;
                    val: TEXT;
                    log : App.Log) =
BEGIN
   IF Text.Empty (val) THEN
      IF db.get(name, email) THEN
         email := "";
      END;
   ELSE
      EVAL db.put(name, val);
   END;
END SetEmail;


PROCEDURE Default (<*UNUSED*>x: HTTPControlValue.TextValue;
                   <*UNUSED*>log: App.Log) =
BEGIN
END Default;
```

The variable `root` defines the root of the HTTP server. `db` is a text-to-text table for mapping names to email addresses.

```
VAR
   root : HTTPControl.StaticForm := HTTPControl.RootForm();
   db := NEW(TextTextTbl.Default).init();
BEGIN
```

Initialize root default options.

```
   root.hasSubmitButton := TRUE;
   root.title := "Contact Database";
```

Add a title.

```
   root.addValue(NEW(HTTPControlValue.MessageValue).init(
      "\n" & "<H2>Contact Database</H2>"));
```

Add the two text fields.

```
        root.addValue(name_value);
        root.addValue(email_value);
```

Serve at port 80. If there is a problem, report it.

```
        TRY
           HTTPApp.Serve(80);
        EXCEPT
           App.Error => Put ("A problem occured\n");
        END;
    END WebContact.
```

6.7 Interacting with C Programs

Most real programs need to interact with an existing body of code. Since CM3-IDE has provisions for describing unsafe operations, binding programs written using CM3-IDE with C is straightforward. In this section, we will describe two programs that call C code on Unix or Win32 platforms, and a Modula-3 program that is called from C.

6.7.1 Calling C: A Unix Example

In this example, we create an interface for accessing the `getcwd` function from Modula-3.

We then wrap a safe interface around the unsafe layer that calls C. This example only works on Unix, but a similar example can be written for Win32 as you will see later. Of course, if we were to only use Modula-3 facilities, the code could easily be ported.

The basic steps in writing this program are:

1. First, read the Unix man page on "`getcwd`" to get some information about its parameter, and what it does.

2. Interface `Ulib` contains the `<*EXTERNAL*>` Modula-3 signature for the "`getcwd`" function in our project. If we are to call this from client code, we'd have to make that unsafe, and have to deal with C data structures, which is probably not a good idea. So, in the next step, we build a safe wrapper around the C call.

3. Create an interface and an implementation "`Lib`". This will be the Modula-3 wrapper for `Ulib`. The idea here is to create a function, `GetCWD()` which returns a `TEXT` containing your current working directory. `Lib.i3` should be pretty straightforward. All you do is declare the signature of the function.

4. `Lib.m3` is more subtle. What we need to do is allocate some space for the C buffer, and then pass it to "`getcwd`", finally copy the contents of the `getcwd` buffer back into a `TEXT` and return it.

   We can either use `Cstdlib.malloc` for allocating the right buffer, and `Cstdlib.free` to dispose it after copying the buffer into a `TEXT` via `M3toC.CopyStoT`.

5. Create a safe main module and call `Lib.GetCWD()` from it.

The good news is that most of the time, we can program in the safe mode, where the language takes care of things like garbage collection. This eliminates the need for separating safe and unsafe code.

Safe interface. Interface `Lib` provides a safe `GetCWD` interface. This means its implementation must have to deal with bridging from unsafe operations to safe operations.

```
INTERFACE Lib;
PROCEDURE GetCWD(): TEXT;
END Lib.
```

Unsafe implementation of a safe interface. The implementation of `Lib` interface includes the body of `GetCWD`, which calls `malloc` to allocate a string, sends it to `getcwd`, and converts the result to `TEXT` while handing memory management.

```
UNSAFE MODULE Lib;
IMPORT Ulib, M3toC;
FROM Ctypes IMPORT char_star;
FROM Cstdlib IMPORT malloc, free;

PROCEDURE GetCWD(): TEXT =
CONST size = 64;
VAR c_str := malloc (size);
BEGIN
   EVAL Ulib.getcwd(c_str,size);
   WITH result = M3toC.CopyStoT(c_str) DO
      free(c_str);
   RETURN result;
   END;
END GetCWD;

BEGIN
END Lib.
```

Unsafe interface to Unix libraries. Interface `Ulib` defines an external function `getcwd`.

```
INTERFACE Ulib;
FROM Ctypes IMPORT char_star, int;

<*EXTERNAL*>
PROCEDURE getcwd(result: char_star;
                 size: int
                ): char_star;

END Ulib.
```

The main module. The main body of the code is simple, because `Lib` takes care of bridging the safety gap.

```
MODULE CallingC EXPORTS Main;
IMPORT IO, Lib;
BEGIN
   IO.Put (Lib.GetCWD() & "\n");
END CallingC.
```

Makefile. The makefile is quite ordinary.

```
import("libm3")
interface("Ulib")
module("Lib")
implementation ("CallingC")
program ("m3pwd")
```

6.7.2 Calling C: A Win32 Example

In this example, we create an interface for accessing the `MessageBox` function from the Win32 API. To do so, we import the interface `WinUser` which defines the signature of the `MessageBoxA` call. We then call `WinUser.MessageBox` from the main module, `OK`.

This example only works on Win32, since `MessageBox` is a Win32 call. Since this call is not available on other platforms your program is not portable. Of course, if you were to only use the portable interfaces available in Modula-3, you would not have any portability problems.

`WinUser` defines basic Win32 API user-level calls. `M3toC` defines mappings from Modula-3 to C strings.

OK.m3.

```
UNSAFE MODULE OK EXPORTS Main;
IMPORT WinUser, M3toC;
IMPORT Params;
VAR
   message: TEXT := "";
BEGIN
   FOR i := 1 TO Params.Count-1 DO
      message := message & Params.Get(i) & " ";
   END;
   EVAL WinUser.MessageBox(NIL,
                  M3toC.TtoS(message),
                  M3toC.TtoS("A CM3-IDE Example"),
                  0);
END OK.
```

Here is the portion of the `WinUser` interface where `MessageBox` is defined:

```
INTERFACE WinUser;
…
PROCEDURE MessageBoxA (hWnd: HWND;
                       lpText : LPCSTR;
                       lpCaption: LPCSTR;
                       uType : UINT
                      ): int;

CONST MessageBox = MessageBoxA;
…
END WinUser.
```

6.7.3 Calling Modula-3 from C

This example demonstrates how to call Modula-3 procedures from C. The C procedure in the example takes a single parameter which itself is a parameter-less procedure that returns an integer. Have the C function call the passed the procedure, add one to the result and return the new value. The makefile will assume that the C code is in a file named `Cstuff.c`.

```
INTERFACE Cstuff;

TYPE
   IntProc = PROCEDURE (): INTEGER;

<*EXTERNAL*>
PROCEDURE add_one (p: IntProc): INTEGER;
(* Returns "1 + p()". *)

<*EXTERNAL*>
PROCEDURE add_one_again (): INTEGER;
(* Returns "1 + m3_proc()". *)

<*EXTERNAL*>
VAR m3_proc: IntProc;

END Cstuff.
```

```
MODULE CcallsM3 EXPORTS Main;
IMPORT IO, Cstuff;
VAR
   x: INTEGER := 33;
   i: INTEGER;

PROCEDURE Foo (): INTEGER =
BEGIN
   INC (x);
   RETURN x;
END Foo;

BEGIN
   IO.Put ("calling add_one.\n");
   i := Cstuff.add_one (Foo);
   IO.Put ("add_one () => ");
   IO.PutInt (i);
   IO.Put ("\n");
   IO.Put ("calling add_one_again.\n");
   Cstuff.m3_proc := Foo;
   i := Cstuff.add_one_again ();
   IO.Put ("add_one_again () => ");
   IO.PutInt (i);
   IO.Put ("\n");
END CcallsM3.
```

`CStuff.c` calls (and is called by) the Modula-3 code.

```
#include <stdio.h>
typedef int (*PROC)();
int add_one (p)
    PROC p;
{
   int i;
   printf ("in add_one, p = 0x%x\n", p);
   i = p ();
   printf (" p() => %d\n", i);
   return i+1;
}

PROC m3_proc;
int add_one_again ()
{
   int i;
   printf ("in add_one_again, m3_proc = 0x%x\n", m3_proc);
   i = m3_proc ();
   printf (" m3_proc () => %d\n", i);
   return i+1;
}
```

Makefile. The call `c_source` compiles a C program with your C compiler. See the Operations Guide for `cm3` at `/help/cm3/cm3.html` or the `cm3.cfg` file in your installation for more information.

```
import ("libm3")
c_source ("Cstuff")
interface ("Cstuff")
implementation ("CcallsM3")
program ("ccallsm3")
```

6.8 Summary

This chapter described a handful of complete programs to illustrate the use of advanced programming facilities in CM3-IDE. You can find the sources for the programs in this chapter in the  Examples section of your CM3-IDE environment.

Robust Distributed Applications. Network Objects can be used to build robust distributed applications. See the `NetObj` interface for more information (See page 108).

Client/Server Computing. Using the safe TCP/IP interfaces, you can build multi-threaded client/server applications that use the socket interfaces. TCP/IP interfaces abstract away the differences between Unix sockets and Winsock implementations. See the `TCP`, `IP`, and `ConnRW` interfaces for more information (See page 115).

Data Manipulation. Using the `Pickle` interface, you can take snapshot of complex object graphs, and later load them into memory. The `Fingerprint` interface allows you to compare large data structures efficiently. The `IO`, `Rd`, `Wr`, `Lex`, `Scan`, and `Fmt` interfaces help you read and write from I/O streams (See page 120).

Portable Operating System Interfaces. CM3-IDE provides interfaces for accessing operating system services in a platform-neutral manner. See the `Process` interface for managing processes, `File`, `FS`, `FileWr`, `FileRd` for filesystem access, `Thread` for creating new threads and concurrency control, `Params` for command-line parameters, `Env` for environment variables, and `Time` for the system clock. Using these interfaces, you can write portable programs that access various operating system facilities (See page 124).

Dynamic Web Applications. You can build dynamic web applications using the web toolkit. Read the `HTTPApp` interface as a start (See page 135).

Accessing Legacy C code. Binding unsafe portions of your program to C code is straightforward, but tedious. To aid portability and robustness of your application, you should avoid using legacy C code as much as possible (See page 137).