APPENDIX C. TOWARD A DISTRIBUTED, OBJECT-BASED FORTH

C.1 Objectives

As the complexity of the accelerator expert system increased, it quickly became clear that an object-based[1] approach would be useful. For example, all rules may be asserted with a new truth value, but the actions involved to assert this value may vary depending on the particular rule: a simple fact may just involve a store, while a temporal fact may require updating of historical information. The obvious way to manage this is to define classes of rules with associated methods.

An object model is also well suited to distributed processing. The concept of "send a message to an object" translates directly to a network message with parameters, whereas a remote procedure call involves the further (and subtle) problems of blocking and parameter return. A message passing protocol may be implemented with nonblocking calls, greatly preferred for this application.

This distributed object implementation should meet four objectives:

a) *Late binding*. Processor A may not know the class of the rule on Processor B, to which a message is being sent. Thus, late binding is implicitly required.

b) *Fast.* Since methods such as *assert* and *invoke* are used frequently in the real-time control loop, it is important that their execution be fast. This precludes many of the late-binding techniques currently in use (Rodriguez and Poehlman, 1996).

c) *Abstract.* Many early-binding systems achieve speed by precompiling object or method references as addresses in the CPU data space. In a distributed system of mixed CPUs, this technique is not feasible. All objects and methods must be identified by abstract labels or selectors.

d) *Not encapsulated.* For the immediate application, there is no pressing need for encapsulation or

1. (Tanenbaum, 1995) makes the distinction between object-based and object-oriented systems. Briefly, object-based systems use objects and methods, but with no attempt at encapsulation.

information hiding, or for any particular syntax.

C.2 Implementation

All objects in a given CPU are referenced by "object numbers" starting from zero. Within each CPU is an "object index," which points to the data structure for each object (Figure C-1).

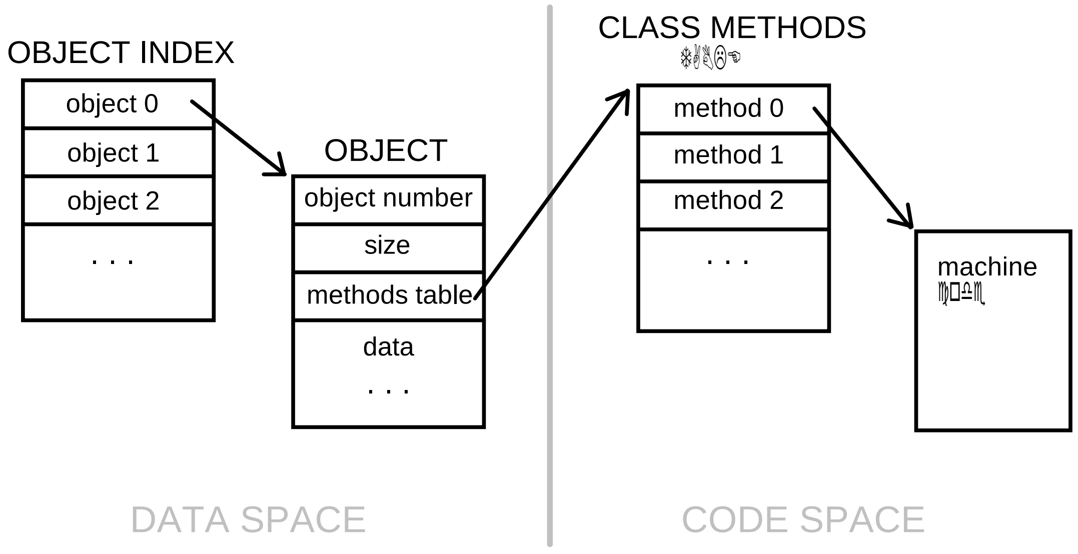

Figure C-1. Object Data Structures

The object data structure contains a header, followed by the instance data for the object. The header includes the object number,[2] size of this instance, and a pointer to the methods table for this class of object.

Like objects, all methods are referenced by method numbers starting from zero, which index into a table. In this case, the table contains the address of Forth words (typically machine code) which perform the actions for each method. Method tables have been previously employed (Morgenstern, 1990; Zsoter, 1995). Method numbers are shared among all classes; e.g., *assert* is method #3 in every class, and classes which do not support *assert* are required to have an *undefined* method in table entry 3. This means that

2. References to "self" need to determine the object *number*, given the object's *address*.

for most classes, the methods table is sparsely populated -- a tolerable tradeoff for execution speed -- and that method numbers must be carefully assigned (to be discussed shortly).

Object data, which can change, must be located in the Data (RAM) space of the embedded processor. Methods are fixed at compile time, and thus located in Code (ROM) space. Once compiled and burned into ROM, the methods of a class are fixed -- but an object may be changed to a different class.

Executing a given method for a given object thus involves an index and fetch into the object table, an offset and fetch from the object, an index and fetch into the methods table, and an indirect jump. This requires only 9 machine instructions on the 8086, and 12 machine instructions on the 68HC16, an acceptable overhead for a late-binding procedure call.[3]

C.3 Syntax

Class data structures (instance variables) are defined using an extension of the common Forth structure declaration word set (Bradley, 1984; Pountain, 1987; Bartel and Bartel, 1989). The syntax for a declaration is

```
INHERIT classname
      size ITEM fieldname
      size ITEM fieldname
       . . .
CONSTANT sizename
```

The data structure[4] of the parent class is inherited, and additional instance variables may be defined with ITEM. If this is a totally new class, it inherits from superclass COMMON. After the last ITEM is defined, the total size of the instance data is available on the stack; this is defined as a normal Forth CONSTANT so that it may be used later in the class definition.[5]

Next, the method routines can be defined. Each method is a Forth word -- high-level or machine

3. (Zsoter, 1996) describes the use of CPU registers to hold the address of the current object, and the address of its methods table. With this technique, the late-binding call overhead can vanish completely on some processors, e.g., the 80386.

4. Strictly speaking, only the data allocation is inherited here. Instance variables are visible to all classes, and so are available to the child class.

5. A CONSTANT is not necessary, but is much safer in the event of a stack imbalance during the definition of the methods.

code -- which will receive the address of the object on the top of stack. Additional parameters passed to the method, if any, will be on the stack under the object address. The method consumes the object address and parameters, and leaves any return parameters on the stack.

Each method is given a name with the statement

```
n METHOD methodname
```

For example, the first four methods of a "fact" object are

```
0 METHOD :build                  \ initialize new rule object
1 METHOD :scrub                  \ reset rule to "unknown"
2 METHOD :invoke    ( -- d )     \ get rule's truth value
3 METHOD :assert    ( d -- )     \ set new truth value
```

These indices refer to locations in a manually constructed methods table, containing the addresses of the corresponding Forth words:

```
CREATE tablename   ' word ,  ' word , ...
```

or, in this example,

```
CREATE FactMethods
    ' FactBuild ,   ' FactScrub ,
    ' FactInvoke ,  ' FactAssert ,
```

The class is then defined with

```
sizename tablename CLASS classname
```

Objects may be instantiated dynamically at run-time, or statically at compile-time. The latter is preferred due to its faster and deterministic run-time speed. Invoking "classname" will cause an object to be created, given a name, and added to the object index:

```
classname objectname
```

When "objectname" is executed, it will then return the object number. Executing a "methodname" will then apply a method to that object. By convention, method (message) names begin with a colon (:), so to the programmer, the message passing syntax is

```
(optional parameters)  object :message
```

C.4 Further Work

No effort has been expended to make the syntax "pretty." A starkly functional package was adequate to develop the object-based expert system, and allows experimentation with the "internals" of the object implementation. An improved syntax would make the code easier to write and to understand.

The greatest current deficiency is the need to manually assign and track method numbers, and manually construct the methods table. This could be automated if a suitable assignment algorithm were employed. (Hamilton, 1996) describes an object system remarkably similar to this one, and suggests the use of a graph-coloring algorithm to efficiently assign method numbers (selectors).

There has as yet been no need for encapsulation or information hiding. All methods, data structures, and selectors are visible at all times. Should encapsulation become desirable, any of the techniques employed by previous object-oriented Forths -- vocabularies, dictionary relinking, or private symbol tables -- should be directly applicable.

At present, only single inheritance of instance data is explicitly supported. (Method inheritance is implicitly supported, since all methods are global and the methods table is manually constructed.) As rule classes have been defined to perform specialized functions (PID control, history tracking), the need for multiple inheritance has become evident.

Although this package builds the foundation for a true distributed object-oriented system, such a system has not yet been implemented. This will probably involve the creation of a "DoMethod" network message, plus messages to pass parameters. Methods currently may return parameters as well as accept messages; either some network mechanism for returned data must be devised (analogous to Remote Procedure Call), or the object-programming model should be revised to eliminate the need for return values (e.g. by returning object messages instead).