```
****************************************************************************
*                                                                          *
*                      the 'talk' communication manager                    *
*                                                                          *
*                          by Geoffrey P. Coco                             *
*                                                                          *
****************************************************************************
```

DEFINITIONS

Grouples, the VEOS all-purpose data elements, also serve as the smallest units of VEOS inter-entity communication.

All VEOS entities have a unique entity id (entity UID) associated with them which is used for runtime entity discrimination. An entity UID is *always* stored and passed as a string.

Conveniently, the entity UID is also the internet address of the machine that the entity is running on - plus the service name that the entity may be associated with - plus the UNIX socket number that the entity is using for incoming communication.

An example of a entity UID is: (callay nil 9000), use string "nil" for unspecified service.

When any VEOS entity is invoked, the entity searches the grouplespace in order to discover it's entity UID. So, the UID *must* be supplied in the entity's .fig file.

'talk' uses entity UIDs to completely and uniquely specify inter-entity communication addresses. That is, an entity's UID is also it's 'talk' address.

MODUS OPERANDI

'talk' is contained within the 'talk' C library and is linked to each entity shell. All entity shells automatically initialize their associated talk code at startup. Thus, all VEOS entities are equipped with inter-entity communication capabilities.

When an entity shell is invoked, it initializes 'talk'. This consists of setting up a private memory cache and opening a connection for incoming messages from other entities.

There are three configurable runtime entry points into the 'talk' library. The entity can send a message (speakTalk entry point), pass temporary control to the incoming message daemon (listenTime entry point), or pass temporary control to the outgoing message daemon (speakTime entry point).

When the entity needs to send a message to another entity, the entity generates a send message request to 'talk' (via speakTalk) in the two ways described below. If there is already an outgoing connection established to the specified entity and no messages are currently queued for that destination, speakTalk sends the message directly. Otherwise, speakTalk establishes and confirms the connection and queues the message for that entity destination.

Then, when the outgoing message daemon is allotted a slice of processing time (via speakTime), 'talk' completes all pending outgoing message requests to all entity destinations.

Similarly, when the incoming message daemon is allotted a slice of processing time (via listenTime), 'talk' retrieves all pending incoming messages from all open incoming connections.

Each incoming message is a single grouple. As 'talk' retrieves incoming grouple messages, if the grouple contains a primitive, 'talk' dispatches the primitive immediately. Otherwise, 'talk' posts the new grouple to the grouplespace.

When an entity shell terminates gracefully (e.g. user initiated), the shell brings down 'talk'. This consists of closing all connections properly so that entity shells that were connected may survive the change.

HOW TO USE TALK FROM WITHIN A VEOS ENTITY PRIMITIVE

1. Generate outgoing messages by either of the two means described below. Nothing special is required to receive incoming messages, provided rule 4. is followed.

2. When an entity at the other end of an outgoing connection is about to terminate (see entityRegister primitive in how_to_make_a_veos_entity) call speakClose() directly from C with that entity's UID as the only parameter.

3. Link with: shell.o,
   with options: -ltalk.

4. Give ample entity processing time to 'talk' by using 'talk' daemons. Do this by placing the following grouples in the entity's .fig file:

```
                [ $listenTime ]
                [ $speakTime ]
```

HOW TO SEND A MESSAGE - TECHNICAL (two ways)

A. Make function calls to speakTalk() directly from C. speakTalk() takes a variable number of parameters. The parameters should look like:

```
          speakTalk(int,          /* number of parameters to follow */
                  char *,   /* the UID message destination */
                  char *,   /* variable # of message words */
                  ...)
```

```
example 1: { ...
             iSuccess = speakTalk(4,
                            "callay nil 9000",
                            "blah",
                            "blah",
                            "blah");
          ... }
```

This direct call to the C function speakTalk() would cause 'talk' to send the grouple message: [(blah) (blah) (blah)] to the entity at internet address: (callay nil 9000).

-- OR --

B. Post grouples to the grouplespace with first word - ("speakTalkPrim"), second word - (entity UID), and subsequent words - (message data).

example 2: [(speakTalkPrim) (callay nil 9000) (blah) (blah) (blah)]

The presence of the above grouple into an entity's grouplespace would cause
that entity's standard primitive - speakTalkPrim - to send the grouple message: [(blah) (blah) (blah)] to the entity at internet address: (callay nil 9000).

NOTE: each instance of a call to speakTalk() or a (speakTalkPrim) grouple generates a *single* inter-entity message, no matter how many message words are passed.

These two methods of sending inter-entity messages are functionally equivalent except for subtle performance differences...

When speakTalk() is called directly from C, 'talk' gets processor control immediately and can attempt to complete the message send request right away as described above in MODUS OPERANDI.

But, when a grouple is posted which contains the primitive speakTalkPrim, that grouple, a potential message send request, sits in the grouplespace until the primitive dispatcher (part of the VEOS shell) discovers it. Once the primitive dispatcher does find the grouple, it invokes the primitive speakTalkPrim which calls speakTalk() directly from within it's code.

Although this latter method conforms to the general VEOS grouple model, several other entity primitives may execute before the request is even found, thus delaying actual message transmission several entity shell processing cycles (0 - 2 seconds, depending on the entity processing load).

Furthermore, if it is sure that the entity will be terminating after the next shell processing cycle, it is safer to call speakTalk() from C to assure that the message is sent before termination.

HOW TO SEND A MESSAGE - PHILOSOPHICAL

As is emphasized in how_to_make_a_veos_entity, primitives have a fundamental obligation to the world to be hasty and efficient (self-enforced time sharing). To be sure that primitives uphold this obligation with respect to inter-entity communications, keep these tenets in mind.

1. Do not place importance on whether a sent message arrived at the destination. In other words, acknowledgement and send-reply paradigms are highly discouraged. Two reasons:

First, VEOS primitives perform network duties at or above the session layer of the ISO network model. Error-free transmission is the duty of lower level layers. For VEOS, error-free transmission is ensured by unix tcp protocol and by 'talk' - not by entity level code.

Second, and more important, in order to simplify deadlock issues, 'talk' was designed for half-duplex protocols only. 'talk' purposely does not provide a mechanism for an entity to suspend itself until a message arrives - a full-duplex feature. The result is that 'talk' provably excludes the hold-and-wait deadlock condition from ever occurring.

NOTE: Two-way transmission between two entity shells can be achieved with two half-duplex connections in opposite directions.

Consequently, it may take considerable time for a message to reach it's destination and undergo processing before a return message is transmitted.

Return messages may lag between 10,000 ms and 2 seconds or more from transmission time.

2. Send no more than one message per destination per entry into a primitive. This allows for easy world calibration, and network sanity. The destination entity can only process one message per cycle (average, depending on entity load, etc.).